

The AmigaWorld

TECH JOURNAL

•PREMIERE ISSUE•

ARTICLES

- 4 **Menus for a New Generation**
The user comes first in 2.0 menus
- 9 **Control Your Channels**
Delve in the audio hardware
- 13 **Recycle Your Sprites**
Reuse sprites vertically
- 26 **Probing Your Current AmigaDOS Device List**
What's active on your system?
- 30 **Shared Libraries for the Lazy**
Write them once, use them often
- 32 **The Fast Floppy System**
Explore the newest file system
- 34 **In Search of... The Perfect Joystick Routine**
The wrong way may be right
- 36 **An Amiga Basic Grapical Interface**
More clicking, less typing
- 38 **The Amiga Zen Timer**
Optimize your code
- 43 **Do-It-Yourself Software Marketing**
Code it, package it, sell it

COLUMNS

- 1 **Message Port**
A welcome and a promise
- 7 **Digging Deep in the OS**
Make 1.3 programs look their best in 2.0
- 16 **Graphics Handler**
An explanation of the deep-bitmap standard
- 23 **TNT**
News and products from the technical community
- 48 **Letters**
What's new with you?

REVIEWS

- 20 **CygnusEd Professional 2.11**
The tried-and-true text editor just got better
- 21 **Macro68 and Algebra**
New assemblers, new formats
- 40 **The QPMA BASIC Programmer's Toolkit**
Graphics help?
- 41 **FlexeLint 4.0**
It finds the bugs, you fix them

ON DISK: (See page 25)

DICE: Dillon's Integrated C Environment

PowerWindows 2.5c demo

Plus source and executable code for articles





*Captain's Log 217:2055
Time shift equalization routines recoded in
Aztec C. All systems go. It's good to be back home.*

New Release Aztec C 5.2

- Amiga
- Macintosh
- MS-DOS
- 80x86, 680x0 ROM

Call Today for
Special Discount Prices
and Technical Information

Space Station 2055 - Viido Polikarpus

Aztec C is available for Amiga, Macintosh, MS-DOS, Apple II, TRS-80, and CPM/80. Native and cross development.

Aztec Embedded C hosts on MS-DOS and Macintosh. Targets include: 68000 family, 8086 family, 8080/Z80/64180, & 6502

Aztec C includes - C compiler, assembler, linker, librarian, editor, source debugger, and other development tools.

MANX 800-221-0440

• Outside USA: 908-542-2121 FAX: 908-542-8386 • Visa, MC, Am Ex, C.O.D,
domestic & international wire • one and two day delivery available

Manx Software Systems • 160 Avenue of the Commons • Box 55 • Shrewsbury NJ 07702

MESSAGE PORT

Finally, The First

AT LAST. *The AmigaWorld Tech Journal's* premiere issue is done and in your hands. Since 1987 when the idea for a technical journal was first proposed, I've feared it would forever be buried in the "Good Ideas That We Ought To Get Back To Someday" pile. More persistent than most, the tech journal proposal kept bubbling its way to the top of the stack. Each stint at the top refined the idea a bit more. By last fall, the once hard-core, professional-developers-only newsletter had expanded into a magazine for all technically inquisitive and serious Amiga users. The two most exciting and important enhancements were the founding of the Peer Review Board and the addition of a companion disk.

Staffed by engineers and professional programmers, the Peer Review Board is dedicated to ensuring the accuracy of all articles and keeping *The Journal's* technical level high. Graphics specialists read the "Graphics Handler" column, the operating system developers read "Digging Deep in the OS," Commodore hardware designers and CATS members make sure the system specs we print are the system specs they created. Peer Reviewers check the text and test all associated code thoroughly, primarily concentrating on the facts, but also considering the article's solution to the presented problem. We want to teach good programming practices, not just provide keen routines. If a listing's approach is a too roundabout, we'll make the author untangle the snarl of gotos, reconsider, and rewrite.

To minimize "Yes, it's perfectly fine, but my code's better" arguments, each article is read by two reviewers. This

way, when code wars do arise, the four-way discussion usually results in new article ideas and more information for the reader, instead of a standoff.

(If you're saying "I could do that. I know as much as these guys, probably more!" maybe you should be giving me a call.)

While the Peer Review Board improves *The Journal's* quality, the inclusion of a disk with every issue expands its range. No longer do we have to turn down valuable articles because the code would fill three-quarters of our pages. Instead, I just copy the source and executable files to a disk, and hand it to Mare-Anne Jarvela, our manager of disk products and jigsaw-puzzle whiz, who compiles the disk you'll find on page 25. Any leftover space she quickly fills with utilities and PD languages. Best of all, you don't have to retype the routines you need only to spend hours and hours hunting for that lost semicolon. Hurray!

BUT WHAT WILL WE READ ABOUT?

While we aim to help you become a better programmer and more technically aware, we're not going to waste your time with three-part dissertations on C's printf() command or how to hook up your VCR. We will teach you how to better use the Amiga's unique capabilities—which ROM Kernel routines to call when, how the hardware works, which graphics algorithms to use, and so on. By covering all the major languages, we'll help you make the jump from BASIC to C, or C to assembly, or tie everything together with ARexx.

I can't promise Technicolor, Cinemascope, surround sound, or the famous "Cast Of Thousands," but here are a few of *The AmigaWorld Tech Journal's* coming attractions:

- Using CreatePort() and DeletePort() to write an input handler.
- 3-D graphics algorithms and implementations

- Handling HAM
- C compilers: SAS versus Manx
- A roundup of time-base correctors
- A thorough examination of the NTSC/RS170A standard
- Controlling/monitoring IPC (inter-process communications)
- Writing an Anim player
- Inside ARexx, ARexx and the Toaster, ARexx and you name it
- An explanation of the MIDI standard
- Taking advantage of Super Denise's new genlock modes
- Programming Unix
- Working with object-oriented programming and C++
- Understanding outline font technology

Waiting for a topic that you don't see? Let's hear from you then! I'm always open to your suggestions.

During the the Amiga's dark ages when few people had even heard of the machine, let alone *used* one, the techies loyally stood behind it, educating anyone who would listen. Now the bandwagon is finally filling up with neophytes boasting "I always knew the machine would just take off one day," you can smile and say, "Ah, but I worked with her when. . ." *The AmigaWorld Tech Journal* is here to recognize that dedication and foster your new innovations, a professional publication for a professional machine. At last.

Linda J. B. Laflamme

Linda Barrett Laflamme, *Editor*

Louis R. Wallace, *Technical Advisor*

Barbara Gefvert, Beth Jala, Peg LePage, *Copy Editors*

Brad Carvey
Joanne Dow
Keith Doyle
Andy Finkel
John Foust
Jim Goodnow
Scott Hood
David Joiner
Sheldon Leemon
Dale Luck
R.J. Mical
Eugene Mortimore
Bryce Nesbitt
Carolyn Scheppner
Leo Schwab
Steve Tibbett
John Toebes

Peer Review Board

Mare-Anne Jarvela, *Manager, Disk Projects*

Laura Johnson, *Designer*

Alana Korda, *Production Supervisor*

Debra A. Davies, *Typographer*

Kenneth Blakeman, *National Advertising Sales Manager*

Barbara Hoy, *Sales Representative/Northeast*

Michael McGoldrick, *Sales Representative/
Midwest, Southeast*

Heather Guinard, *Sales Representative/Partial Pages,
800/441-4403, 603/924-0100*

Meredith Bickford, *Advertising Coordinator*

Giorgio Saluti, *Associate Publisher, West Coast Sales,
415/363-5230
2421 Broadway, Suite 200, Redwood City, CA 94063*

Wendie Haines Marro, *Marketing Director*

Laura Livingston, *Marketing Coordinator*

Margot L. Swanson, *Customer Service Representative,
Advertising Assistant*

Lisa LaFleur, *Business and Operations Administrator*

Mary McCole, *Publisher's Assistant*

Susan M. Hanshaw, *Circulation Director, 800/365-1364*

Pam Wilder, *Circulation Manager*

Lynn Lagasse, *Manufacturing Manager*

Roger J. Murphy, *President*

Bonnie Welsh-Carroll, *Director of Corporate
Circulation & Planning*

Linda Ruth, *Single Copy Sales Director*

Debbie Walsh, *Newsstand Promotion Manager*

William M. Boyer, *Director of Credit Sales & Collections*

Stephen C. Robbins, *Vice-President/Group Publisher*

Douglas Barney, *Editorial Director*

The AmigaWorld Tech Journal (ISSN 1054-4631) is an independent journal not connected with Commodore Business Machines Inc. *The AmigaWorld Tech Journal* is published bimonthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is \$69.95, Canada and Mexico \$79.95, Foreign Surface \$89.95, Foreign Airmail \$109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603/924-0100. Entire contents copyright 1991 by IDG Communications/Peterborough, Inc. No part of this publication may be printed or otherwise reproduced without written permission from the publisher. **Postmaster:** Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings, and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

CHARTER
OFFER

A source of technical information for the serious Amiga professional.

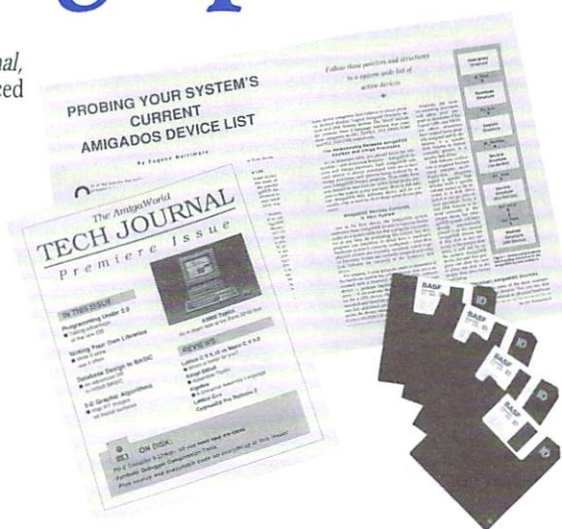
Introducing *The AmigaWorld Tech Journal*, the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bi-monthly issue is packed with fresh, authoritative strategies and advice to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for Public Domain programming on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include some technical information, but none devote every single page to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.



Every issue includes a valuable companion disk!

And only *The AmigaWorld Tech Journal* boasts of a technical advisory board comprised of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

Plus! Each issue comes with a valuable companion disk, including executable code, source code

and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

In every issue of *The AmigaWorld Tech Journal*, you'll find...

- Practical hardware and software reviews, including detailed comparisons, benchmark results and specs
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks
- Programming utilities from PD disks, bulletin board systems and networks
- Wise buys in new products—from language system upgrades to accelerator boards to editing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks. All at special Charter savings. Call 1-800-343-0728 or complete and return the savings form below—today!

To order, use this handy savings form.

Charter Savings Form

☒ **Yes!** Enter my one-year (6 issues, plus 6 invaluable disks) Charter Subscription to *The AmigaWorld Tech Journal* for just \$59.95. That's a special savings of \$29.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal*, I'm entitled to receive a full refund—no questions asked!

Name _____
Address _____
City _____ State _____ Zip _____
☐ Check or money order enclosed. ☐ Charge my:
☐ MasterCard ☐ Visa ☐ Discover ☐ American Express
Account No. _____ Exp. Date _____
Signature _____

Satisfaction Guaranteed!

Or your money back!

Canada and Mexico, \$74.95.
Foreign surface, \$84.97.
Foreign airmail, \$99.95.
Payment required in U.S. funds drawn on U.S. bank.

Complete and mail to:
The AmigaWorld Tech Journal

P.O. Box 802, 80 Elm Street
Peterborough, NH 03458

T491

For faster service, call toll-free
1-800-343-0728.

Menus For A New Generation

By David "Talin" Joiner

A POLISHED INTERFACE can make or break a program's success. Remember, users may spend all day long looking at a single program, and they are liable to shy away from an application with hard to read or cryptic menus, no matter how many functions it has. When properly implemented, menus can be visually pleasing and intuitively organized, and should adapt to the user's customized environment. While what constitutes "visually pleasing" is a matter of taste, outlining how to ensure that menus conform to user-specified options is easy.

GIVE THEM AN OPTION AND THEY'LL TAKE A MENU

One of the biggest changes in Amiga OS 2.0 is that the user gained much more control over system appearance. For example, in 1.3 and earlier, to get a CLI to use a font other than Topaz requires a substantial bit of hackery. Under 2.0, however, changing the font requires only a few mouse clicks in a Preferences editor. Two of the most obvious attributes that affect the appearance of menus are fonts and screen colors.

Fonts: Menu appearance is determined greatly by the text's size and typeface. Many applications have a hard-coded font and hard-coded menus. Numerous others use the default font, but foolishly assume that it will always be Topaz 8. The result is an unreadable mess. Smart applications use the system's default font (which the user can change) and adjust their menus accordingly.

When a program draws a menu item, it first checks in the font pointer of the `IntuiText` item attached to the menu. If the font pointer is `NULL`, then the program looks in the screen structure for the screen font. (Note that the menu headers on the menu bar always use this pointer, because they have no place to put a font pointer.) The screen's font pointer is set when the screen is first opened. If the screen has not been informed what font it should use, it defaults to the global `ScreenFont` pointer. In 2.0, the user can set the screen font via the Fonts Preferences editor.

For many applications all you need to do is make sure that the screen font pointer is set to a known font and then hard-code the menu item positions based on that font's size. I strongly recommend, however, that you go the extra distance and use the system screen font, adjusting the menu items to fit. The reason is simple: With some of the new high-resolution graphics modes (or a visually-impaired user), those small fonts may be completely unreadable.

Colors: Because the default palette is different under 2.0, menus appear dark instead of white (for the reasons behind

this, see the "Digging Deep in the OS" column). I do not advise trying to change this color scheme, because the Amiga A symbol is a hard-coded image in the Kickstart ROM and cannot be changed. If the window's `BlockPen` is set to anything other than 1, the character will look ghastly. (This is also true under 1.3.)

Another consideration is that some users are going to be in monochrome mode. Because the new Productivity mode takes up a lot of system bandwidth, users who want the extra resolution but not the speed penalty may drop down to a single bitplane. To be safe, make sure that all your menus are easy to read on a two-color screen. The new `DrawInfo` structure can be a help here; it has an array of pen colors that you can use as advice when drawing such things as dropshadows, text, and so on.

Applications that open their own custom screen are not quite so constrained; they do not have to keep the colors in the same order as the Workbench screen. The `DrawInfo` structure allows remapping of all the pens used by the system imagery.

To further help you conform your menus (and more) to the new look of Amiga OS 2.0 Commodore is working on an extensive set of user interface design standards, which is scheduled to be published in the spring. This comprehensive document will help application writers design their programs to share a common look and feel with other popular Amiga programs, making all of them more familiar and easier to learn.

ELEMENTAL CONCERNS

With the broader changes clarified, let's make a closer inspection of a menu's individual features and see what you must do to make them functional and attractive.

Text Items: Include some white space around the text. I usually make the menu highlight area three to four pixels taller than the actual font and center the text in the highlight area. Normally, I do not take the area below the font baseline into account when centering, but that's just my personal preference. As long as all the highlight boxes are the same width they should look fine. Make the highlight areas wide enough to just touch one another, but not overlap. This is especially important for submenus; if the highlights overlap annoying flashing will result.

Images: Only a very few programs benefit from using images rather than text in menu items, usually for such options as line width or line pattern selection. I recommend that you avoid using images to represent commands in menus; menus that use images tend to be incomprehensible.

*Attention to user-defined details
and use of the GadTools library will ensure that your menus are
functional and attractive under 2.0.*

Command-key Symbol: This symbol appears whenever a menu item has a keyboard equivalent. Unlike the checkmark, the command-key symbol cannot be changed. It is always blitted in as a rectangle with no masking, and the four corner pips are rendered in Pen 1. For visual continuity, therefore, the menu box background must be rendered in Pen 1, which is controlled by the BlockPen of the window.

Requester Indicator: According to the Amiga user-interface standards (both the old ones in the *Amiga ROM Kernel Reference Manual: Libraries and Devices* [Addison-Wesley] and the new ones), any time a menu item represents a feature that would cause a temporary interruption in the flow of input (such as displaying a file requester) the text for that menu item should end with an ellipsis (...). This is a standard clue that the user will be presented with more choices and will have to do additional clicking and selecting to return to the original location.

Submenu Indicator: A second standardized clue, the submenu indicator is the Alt-0 character, which looks like two tiny greater-than symbols (>) pressed together. Whenever one of your menus has submenus, put a submenu indicator on the right edge of the menu highlight box. If the menus are in a proportional font, adjusting the spacing so that the symbol appears exactly at the right edge may be difficult.

You can get around this by linking to the first IntuiText structure a second one that contains only the Alt-0 character. You can then adjust the symbol's position down to a pixel. An added advantage of this method is that all the items within the same menu heading can use that second IntuiText structure, because the spacing relative to the menu highlight box will be exactly the same for each item.

Accelerator Keys: These are command keys that do not require the user to press the Amiga key. (Electronic Arts' DeluxePaint III, for example, has many of these.) Accelerator keys usually appear in the menu on the right side of the highlight box. Sometimes a modifier key will also be listed, as in Alt-A. The right justification techniques for the submenu indicator apply here as well.

Separator Bar: A new feature is the separator bar, a line of dots that separates the menu items into logical groups. (You can see one in the Icons menu of Workbench 2.0.) The separator bar is an actual menu item, although it is disabled so that it will not highlight. The imagery for the standard separator bar is really a two-pixel-thick horizontal solid line that has then been ghosted because the menu is disabled. You can fake one of these easily using an image and can ar-

range it so that you need only one such image per menu header.

ALL IN THE ROUTINE

Having an application individually adjust the position of each menu, menu item, and associated symbols would be far too tedious and inefficient. The solution is to create an automatic menu layout routine. First you create a set of structures describing the desired menus (these structures may or may not be actual Intuition Menu structures), then lay out routines that use these structures to build a linked list of menu structures to attach to the window. This works well in practice, as long as you are willing to make certain assumptions. Most applications use less than one third of the possible features that menus provide, so you can greatly simplify the design of the layout routines by assuming that the menus will be relatively homogenous.

Amiga OS 2.0 has a set of layout subroutines ready-made as part of the *GadTools* shared library. This system works splendidly, but only under 2.0. If your application needs to work under both 1.3 and 2.0, it will require a its own layout routines. Fortunately, designing them is not hard; I suggest using *GadTools* as a model.

The problem of writing a layout engine can be divided into two parts: creating the menus and laying out the menus. The reason for dividing it this way is that some applications have menus that change based on the mode they are in. You do not want to re-create all the menus every time the mode switches happens; you just want to reposition them to accommodate the changes. If the program is a quickie, you can do away with the creation part altogether and supply ready-made menu items that are modified by the layout phase. This may be tedious, however, if the menu has a lot of subitems and accelerator keys.

The creation phase is fairly simple. Allocate a MenuItem structure for each text item or separator bar, plus an IntuiText (and associated text buffer) for each text label, accelerator key, and subitem symbol. Apply some cleverness, and you will need to create only one subitem symbol for each menu header. The program must also keep track of these items so that it can deallocate them on exit. One way to do this is to walk the list of menu items, freeing any structures encountered.

The laying out procedure is somewhat more complex, but still fairly straightforward. The first thing the program must do is gain access to the screen structure. If the window is already open, then your program can just look in the Window->WScreen pointer. If the application is using a custom screen with a hard-coded font, then it can just use that. On the other ►

hand, if the window is going to open on the Workbench screen or on a custom screen that shares the same attributes as the Workbench screen, then your program must gain access to the Workbench screen. Under 1.3, `GetScreenData()` should be used, while under 2.0 `LockPubScreen()` is preferred. Write the code so as to choose between these two methods based on the version of the operating system.

Once a Screen structure (either Workbench or Custom) has been snagged, you can figure out which font to use and the screen's size and depth. This information is passed to the layout routine so that it knows how large the text items are.

Next, you must lay out the menu headers. If you do not have enough room on the screen to lay out all the headers, the routine might choose to either drop items or abort entirely, returning a failure result.

Now, the layout routine goes through each menu. On the first pass, it calculates how big each item has to be to hold all its elements (the text label, the command-key symbols, sub-menu symbols, and so on) and keeps track of the size of the largest item. On the second pass, it makes all the items the same size as the largest and sets their vertical spacing based on whether the item is a text item or a separator bar. Also, if the menu header is near the right edge of the screen, the items will need to be offset to the left so that they do not run off the right screen edge.

You must also check that the menu does not have so many items (or use so large a font) that the list extends below the bottom of the screen. This not only looks bad, it can cause Intuition to crash with gorgeous fireworks. The solution is to drop down to a smaller font and start the layout all over again or to make multiple columns of text menu items. GadTools uses the latter strategy. (In case you were wondering, Intuition supports a maximum of 31 menus with 63 items per menu and 31 subitems per item.)

After the program positions the highlight boxes, it must place the text items in the boxes. Text and symbols should be centered vertically. Text items should be left-justified, while symbols are right-justified. (Ignore command keys at this stage; they are provided by Intuition.) Now all you need to do is attach the menu strip to the window and start collecting events. (For an example of a simple layout engine, see `example1.c` in the Joiner directory on the accompanying disk.)

As mentioned earlier, Amiga OS 2.0's GadTools library provides ready-made layout subroutines. Fairly simple, GadTools menus work quite well and do not require any special changes to the input event loop (unlike the gadgets part of GadTools). The GadTools routines do pretty much what was described in the previous section, except that they do not currently support accelerator key symbols.

To create a menu, start with an array of *NewMenu* structures. The *NewMenu* structure is much smaller than a normal *MenuItem* structure, but it contains all the fields you really care about in terms of menu definition. It does not contain exact coordinates for the hit box, hidden fields used by Intuition, and a lot of other scary stuff. For example, it offers no field to point to the subitem list; subitems are defined by their type and positioning in the array, rather than explicitly through pointers. (You can see this in the source code for the second example, `example2.c`.) Another nice thing is that if a second window is opened, you can easily create another set of menus from the same *NewMenu* array (you cannot have two windows sharing the same menu strip unless you are looking to get Excedrin Headache #81000003.00217A3E).

To create the real menus, call `CreateMenusA()`. The A stands for the fact that this is a *varargs* routine, meaning that it takes a *TagList* of extra parameters. (A word of explanation: *TagLists* are new in 2.0, and are used all over to implement library functions with variable numbers of arguments. They are also totally beyond the scope of this article.)

After the menus are created, call `LayoutMenusA()` or `LayoutMenuItemsA()`, depending on whether you require layout of a whole menu strip or just a single menu. At this point the menus are ready to be used like normal menus.

With all this algorithmic generation of menus going on, a surprising possibility arises, that of *user-defined menus*. I have one application that reads in a configuration text file at boot time that contains (among other things) all the text and bindings for all the menu items in the program. I can rearrange the menus or convert them to a foreign language simply by editing a text file. Given a basic parser and an extended menu structure like the one described earlier, this is fairly easy. It is just a matter of parsing the label of the menu and name of the function, using that name to look up the address of the function in a table, and then stuffing the function address into the appropriate field. I leave minor details such as checkmarks and mutual exclusion as an exercise for you.

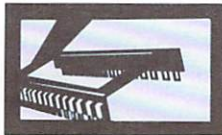
GET TO WORK

After the menus are set up, your program must respond to menu events. The required standard input event loop is familiar to all Intuition programmers, but contains an oft-overlooked subtle point: the use of the *NextSelect* field. Intuition allows the user to make multiple menu selections using the left mouse button while the right mouse button is held down. In these cases, the menu hits do not come in as individual messages, but rather as a single message with all the menu items linked together. To catch all the menu hits, your program must walk through this chain. Note that if a menu is hit twice, it only appears in the chain once. Also, if the user happens to hit the menu again very quickly, while the old chain is still being processed, the chain may be disrupted. Fortunately, this is not a problem because the *NextSelect* field is an index rather than a pointer, so the worst that can happen is a menu selection could be lost.

One innovative technique for making menu processing easier is to use an extended menu structure, which adds extra fields that the application can use at the end of the *MenuItem* structure. In fact, GadTools does this already, by adding an extra field for application use every time it creates a menu item. For example, every menu item could contain a function pointer and have the main event loop call that function whenever the menu is hit. For example, the New File menu would point to the New File function. Say good-bye to complicated case statements!

In the final analysis, empowerment of the user is what it's all about. Part of this process is leaping ahead of the user's expectations, part is conforming to the user's decisions. Responsiveness to the user's preferences regarding system-wide typefaces and colors in your menus is part of this philosophy. Your challenge is to write programs with aesthetic judgement and engineering elegance. ■

David Joiner is the author of Faery Tale Adventure and Music-X, plus an artist, award-winning costume designer, and user-interface expert. Contact him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458, or on BIX as talin.



DIGGING DEEP IN THE OS

Looking Good In 2.0

By Andy Finkel

SINCE THE RELEASE of Kickstart 2.02, unless you are playing strange tricks, odds are that your 1.3 program runs perfectly fine under Amiga OS 2.0. It may not look fine, however; colors may be off, fonts may look strange, or other visual defects may be apparent in normal operation of your program.

Each change we at Commodore made to the operating system was made for a reason, to fix a bug or to add a feature. Unfortunately, your program may have been depending on the exact behavior of a particular bug or feature of 1.3 for correct operation. In general, the problems you will face under 2.0 are those created by our removal of limits on the user's choices. Via the expanded Preference editors, users now have more control over the system setup and environment and can change many more settings than under previous versions of the OS.

As a result, you can no longer depend on the system defaults to be constants; your program has to adapt to user selections. Of course, you could always take the easier route and completely ignore the user's choices in favor of your own hard-coded defaults. The first approach, however, is more rewarding in the long run. Users like to see programs that obey their choices.

The appearance of your product gives the user his initial impression of the product, and in many cases of the Amiga itself. If your program muffs its handling of fonts, colors, the Workbench screen, Intuition rendering, and the cursor, it may never get another chance. Users notice these things right away and do not like them. Attention to these details may mean the difference between winning a customer or losing a sale to a similarly featured but more attractive competing product.

FONT FIXES

Under 1.3, fonts were easy to handle and hard to change. You could almost count on having one of two fixed-width (mono-spaced) fonts by default: Topaz 8 or Topaz 9. Under 2.0, you are more likely not to have Topaz as your default font. You may even have a proportional font as your default screen-text font.

You can solve the problem in several ways. Ignoring the user's wishes, you can explicitly set the font for your window or screen to one your program can handle, such as Topaz 8. A more desirable method is to check which font you get when you open the screen or window and automatically adjust to that font.

This can be difficult, if, for instance, you have a complex screen layout or cannot handle proportional fonts. In this

case, Commodore suggests reverting to the system default text font, which is guaranteed not to be proportional, if at all possible. Only if you have no choice should you revert to a hard-coded default of your own. (GfxBase->DefaultFont contains the user's choice of default mono-spaced fonts. If you can't handle what you are given, look in this location. If you can use the specified font, do so; otherwise, drop to a default of your own.)

Handling fonts properly is not very difficult. The only incorrect choice, really, is to take whatever comes by default and fail to adapt to it. (See "Menus for a New Generation" for some specific suggestions.)

CYCLED COLORS

You will notice that we have changed the default color scheme in 2.0, as well as which color registers contain the light colors and which contain the dark. We redecorated on the advice of top computer fashion designers: Gray is in for this year and blue is out. Gray is sleek and businesslike. Gray is the coming thing. While a few may make fashion faux-pas, I do not expect any programs to be broken by this. After all, the user has always been able to modify the default Workbench colors. The register change, however, requires a bit more explanation.

Under 1.0, 1.1, 1.2, and 1.3, the colors were a very saturated blue background, white text, black, and orange (highlight colors). When we made the decision to move to the more professional background color (gray), the result was white text on a gray background. So that the text showed up better, we reversed white and black in the color registers, ending up with (in order) gray, black, white, and light blue. This simple decision has several important implications, especially when you consider 2.0's three-dimensional look, which depends on the proper color relationships to give the illusion of depth.

Unless your application specifies its own color scheme (difficult on the Workbench screen), under 2.0 your 1.3 application will display white where there used to be black and vice versa. If a 1.3 program creates a 3-D illusion with its own color scheme, this change will reverse the look. Icons, of course, will look somewhat different, to put it mildly. How your application handles these problems depends on whether or not it opens a custom screen.

If you do open a custom screen, you already have a considerable amount of control over your resolution, bitplane depth, and so on, and can use this control to make your application look good under any OS version. The problem is that Intuition would like some additional information that it ►

did not require under 1.3. It would like to know which of your colors you want it to use for text, for shadowing, and for accent to give you the 3-D look. Without this information, Intuition gives you a "mono-look" (very flat) set of system gadgets.

When opening a custom screen using the OpenScreen() call, therefore, make sure you pass an Extended Screen structure, which contains your choices for the screen pens Intuition uses for shadow, text, and shine pen colors. Your gadgets will then be properly rendered as well. You can also use these pen colors for your menus and your menus IntuiText structure, so you can have your menus rendered as dark text on a light background. Because you control the number of bitplanes on your custom screen, you are also spared some of the considerations that an application opening on Workbench has to face.

When on the Workbench screen, your icons are hard to handle: While recoloring an icon in your program is easy, designing an icon that looks good in both color combinations is more difficult. Because the user can specify the number of bitplanes to use, your program must adjust to handle one or four bitplanes in addition to the usual two. You do not have much choice here; if you open on the Workbench screen, you have to handle its number of bitplanes. You can use the standard methods of determining your environment, of course, but it is up to your program to adjust. Do not hard-code the number of colors; if you get fancy with scrolling routines, make sure that they can adapt to changing conditions. By the same token, do not use GetScreenData() when

opening a screen that is just like the Workbench screen. This routine does not give you enough data. Instead, use the new 2.0 calls.

PERFECT INTUITION

Intuition rendering has not been spared changes, either. Intuition used to be pretty relaxed about programs using its private screen real estate, such as window borders, string gadgets, and title bars. While it was not really supported, Intuition let you get away with a lot in terms of programs putting elements in the borders, gadgets in the title bar, and drawing into string gadgets. As of 2.0, Intuition aggressively refreshes the screen area it owns, tending to erase or cover things that wander into borders. You *could* experiment to find the new limits of what you can slip by Intuition, but instead you should rearrange things so they are not in Intuition's private preserve. Taking the time to do so now will help make future compatibility easier to maintain. (In reality, Intuition goes to a great deal of trouble to simultaneously render what it must and to avoid making your program's elements look bad. The easier you make it for Intuition, however, the better.)

CURSES, TWO CURSORS

While the console.device has changed greatly for 2.0, (growing a character map, switching refresh modes, supporting cut and paste), those alterations will probably not cause your program problems. (I know I'm tempting fate saying this. Someone is probably already writing in with a counter-example.) What may cause your program trouble is the use of the cursor under 2.0.

To avoid problems, follow a simple rule: If you are using the console.device and do not want to see the console.device's cursor, *turn the cursor off*. You would be surprised at how many times this rule was not followed in the past. The problem shows up in two places.

Under 1.3, a bug in the console.device gave no cursor on SuperBitMap windows. To sidestep this, several programs created their own cursor, which was all well and good, except the programs forgot to turn off the console.device's cursor. When we fixed the bug for 2.0, these programs suddenly had two cursors and looked quite silly.

A second feature of the console.device causes a similar problem: When a console window becomes inactive, the cursor is ghosted, which is fine, unless of course the application has erased the cursor by means of a rectfill rather than turning it off. In that case, when the window becomes inactive, a little ghost cursor appears. (A kludge was later added to prevent this in most cases, because it was a very common mistake.)

While I have discussed only the common problems that have prevented programs from looking their best under the new operating system, there are many others to keep in mind. We went to a fair amount of trouble to make the system itself look better and more professional. Take advantage of these new features and your programs will benefit as well. ■

Andy Finkel is Manager of Amiga Software at Commodore and head of the 2.0 project. He's been with the company since before the VIC-20 and is one of the people to thank for approving the purchase of the Amiga. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458, or contact him on BIX as afinkel.

commodore AMIGA™

MEMORY CHIP

256K x 4-80 DIP	\$8.00
256K x 4-80 ZIPP	\$9.50
256K x 4-80 (Static for A3000)	\$10.00
1 MG x 8-80 SIMM	\$79.00
1 MG x 4-80 (Static for A3000)	*\$42.95
NEW FATTER AGNUS	\$99.95
Amiga Mouse	\$49.00
External Hi Density 1.52 MG Dr.	\$189.95
1.5 MG Insider Board for A1000	\$299.00
Keyboard for A1000	\$139.95
A2000 Power Supply	\$139.95
Power Supply A500 (Heavy Duty)	\$99.95
4 MG Base Board (for A500 w/2 Mg)	\$249.95
4 MG Base Board (for A500 w/4 Mg)	\$349.95
Amiga Janus 2.0	\$29.95

SUPRA

2MG Expandable to 8 MG (FOR A2000)	*\$210.00
4MG Expandable to 8 MG	\$349.00
40 MG Hard Card	\$549.00
40 MG HD (A500 W/512K)	\$699.00

GYP

40 MG HD for A500	\$699.00
40 MG Hard Card	\$599.00
100 MG Hard Card	\$899.00
A3001 4/0 28 MHz	\$1699.00
33 MHz	\$1995.00

* Special sale this month while supplies last

ASI

Ampex Systems, Inc.

(Not affiliated with Ampex Corp.)

5344 JIMMY CARTER BLVD., NORCROSS, GA 30093

(Orders Only) (800) 962-4489 · Fax (404) 263-7852

(Information & Prices) (404) 263-9190

Control Your Channels

For more flexibility in playing sound waveforms, learn to access the audio hardware directly.

By Jeff Glatt

SUFFICIENT FOR MOST applications, the audio device, with its linked lists, allocation keys, and I/O blocks, is often too slow for musical use or syncing audio to complex animation. In such cases, you can bypass the audio device and deal directly with the underlying hardware.

REGISTER YOUR WAVE

The Amiga has four audio channels, each with five hardware registers. These registers are actual locations inside the Amiga to which programs can only write. Reading them (that is, moving data from them) will cause a system crash.

The first register of each channel holds a pointer to the start of the waveform data (which must be in chip RAM). This register is actually two separate word-length registers that combine to form one register able to hold a long (32-bit address). Called the location register, it contains the address of the point within the waveform at which you want the playback to commence. With it, you can play only an excerpt rather than the entire wave. Be sure, however, that the start address is an even address (word-aligned)!

The second hardware register, the length register, contains the number of words that will be played. The waveform's length should be an even number of bytes, because the audio hardware always fetches two bytes at a time, although it plays each separately. (Each byte of a waveform is an eight-bit linear sample point.) You should divide the number of bytes that you wish to play by two and place the result in the length register. (You divide by two because the audio hardware wants to know how many words, not bytes, to play). The longest sample that the hardware can play automatically is 65,535 words (131,070 bytes) long. To play longer samples, you must change the location and length registers during a channel's "audio-block-done" interrupt (more on interrupts later).

The third register, period, contains the waveform's sampling period from which you determine the waveform's playback rate. The sampling period of a waveform is the inverse of its sampling frequency. If you want to play a wave at the same pitch that it was "recorded" and know the sampling frequency, you can easily determine the sampling period with the formula:

$$\text{period} = (1 / \text{sampling frequency in hertz}) / .279365$$

The constant .279365 is a DMA hardware limitation im-

posed by the Amiga; the system cannot support sampling periods less than 124 (or sampling rates higher than 28,867 Hz). If you subtract from the resulting period value, the waveform will play at a pitch higher than its original. If you add to the period, the waveform will play lower than the original pitch. Unlike with the location and length registers, you can change the value in the period register *at any time* while the channel is playing a waveform. Because the pitch will rise or fall as you decrease or increase the period, you can simulate the pitch wheel on a musical keyboard. You can also simulate vibrato by repeating the following sequence:

1. Raise the period slightly.
2. Delay briefly.
3. Restore the period to its original value.
4. Repeat step 2.
5. Lower the period by the same amount as in step 1.
6. Repeat step 2.
7. Restore the period to its original value.
8. Repeat step 2.

Just make sure that the delays are the same length and the period is always between 124 to 500.

If you do not know a waveform's sample frequency or period, you can still calculate the period register's value. First, visually or mathematically isolate the start and end of one cycle within the waveform and count the number of bytes (sample points) in this cycle. Decide at what frequency in hertz (pitch) you would like to play the wave. Now use the formula:

$$\text{period} = (\text{frequency in hertz} / \text{number of samples in a cycle}) / .279365$$

The fourth register is the volume register, which holds the playback volume. The range of acceptable values is from 0 ►

(the softest) to 64 (the loudest). Like the period register, this register can be changed while the channel is playing. For example, you could lower the volume by one every fraction of a second to slowly fade out a sound, build an entire envelope generator, or achieve a tremolo effect by following the steps for a vibrato and substituting the volume register.

Called data, the final register is where the audio hardware puts the two bytes fetched for output. Usually, the channel is playing back under the supervision of the DMA, so you need not concern yourself with this register. The DMA automatically feeds the next two bytes of the waveform to the channel. When the DMA has sent the number of words that you specified in the length register, the whole process starts again back at the address in the location register. The DMA repeatedly plays your excerpt until you stop the channel.

You do not have to start the channel with the DMA enabled, however, if you prefer to feed the channel two bytes at a time manually. To do so, you would place the bytes (a word with the first sample point as the MSB) in the data register. After the hardware outputs the first word (at the rate specified in the period register), the hardware tells you (via an interrupt) that it wants the next morsel. You better be ready with that data!

While this method is very CPU intensive and generally not recommended, it comes in handy when you need to bypass the DMA's constraints. With the manual method you can store the waveform data in fast RAM, putting the sample points anywhere you want (even out of order). You can also play a waveform larger than 131,070 bytes without double-buffering, because without the DMA the system ignores the location and length registers, letting you decide where to start the sample and how many points to output. No longer held back because the DMA cannot deliver data to the channel faster than 28 KHz, you can directly play a wave sampled at a rate greater than 28 KHz. As long as you are delivering data to the channel, you can set a rate as fast as your code, and the Amiga's digital-to-analog converter (DAC), can handle.

PROPER NAMES

Each of the hardware registers is an address inside the Amiga, and the registers for each channel are adjacent in memory, as shown in the following chart. The chart is set up so that you can place it at the start of an assembly program listing and use the symbolic names.

```
;Channel 0
LOCATION_0 equ $DFF0A0 ;holds a LONG
LENGTH_0 equ $DFF0A4 ;holds a WORD
PERIOD_0 equ $DFF0A6 ;holds a WORD
VOLUME_0 equ $DFF0A8 ;holds a WORD
DATA_0 equ $DFF0AA ;holds a WORD

;Channel 1
LOCATION_1 equ $DFF0B0
LENGTH_1 equ $DFF0B4
PERIOD_1 equ $DFF0B6
VOLUME_1 equ $DFF0B8
DATA_1 equ $DFF0BA

;Channel 2
LOCATION_2 equ $DFF0C0
LENGTH_2 equ $DFF0C4
PERIOD_2 equ $DFF0C6
VOLUME_2 equ $DFF0C8
DATA_2 equ $DFF0CA
```

```
;Channel 3
LOCATION_3 equ $DFF0D0
LENGTH_3 equ $DFF0D4
PERIOD_3 equ $DFF0D6
VOLUME_3 equ $DFF0D8
DATA_3 equ $DFF0DA
```

Before starting a channel, you must *always* initialize the hardware registers. As soon as a channel is started, the system copies the hardware registers to a corresponding set of backup registers. The DMA then uses these backup registers to determine where to find the sample, how many words to play, and so on. Immediately after the DMA transfers the values to the backups, you can safely rewrite the location and length registers. Each channel's audio-block-done interrupt alerts you when the DMA is finished copying to that channel's backups. The audio-block-done is the DMA's way of saying, "I've just finished copying the hardware registers. I've got what I need, so you can change them now." Later, when the DMA finishes playing the number of sample words specified in the channel's length backup register, it accesses the hardware registers again.

Never change the registers while the DMA is copying to the backups. Imagine the problems if you rewrote the location register before the DMA backed it up. The DMA would copy the new waveform's address but the previous wave's length. Who knows what the results would sound like! The only safe times to rewrite the location and length registers are when the channel is stopped (because the DMA will not be fetching data) and immediately up on the channel's audio-block-done interrupt. To act upon the latter signal, you must install an interrupt handler that will rewrite the registers to the new waveform's address and length or signal another task to do so. You need not rewrite the registers, however, if you want the DMA to play the same wave again.

Remember, you can rewrite the period and volume registers at any time; the new values will take effect upon the next data fetch (usually immediately to the human ear). If the next wave's period or volume is different than those for the wave the DMA is currently playing, do not rewrite the period and volume registers at the same time you rewrite the location and length. Because the new location and length do not take effect until the previous wave is finished, you can rewrite them as soon as the DMA copies. Wait until the new wave starts, however, before you rewrite the new volume and period. Your signal that the system has copied the new location and length and is playing the next waveform is when the DMA sets the channel's audio-block-done interrupt again. As long as your program is not in C, your interrupt handler should be able to change to the new volume and period quick enough to avoid an audible glitch. Of course, if the new wave has the same period and volume, you can leave the requesters alone.

One application that may require you to change the location and length registers while playing one waveform is playing an 8SVX file. Most musical waves have a oneShot portion (the initial "sound" of the wave, to be played once only) and a repeat portion (the part that keeps "playing" until you stop the note). You need to set up a channel's registers (before starting the channel) with the address, length, period, and volume of the oneShot portion. When you start the channel, the DMA will copy these values to the backups, start the actual audio output, and set the channel's audio-block-done interrupt. At

this point, Exec calls your interrupt handler. The handler can now rewrite the location and length registers (while the oneShot is playing) with the address and length of the repeat portion. As soon as the oneShot is finished, the DMA will copy the new location and length (thereby achieving a smooth, continuous transition to the repeat portion), and set the channel's audio-block-done interrupt.

The system now calls your interrupt handler a second time. At this point, the repeat has just started playing, so the handler could adjust the period and volume. You do not need to do so, however, because the 8SVX repeat segment should have the same period and volume as the oneShot. For all the subsequent audio-block-done interrupts, you can leave the parameters alone, simply not change anything. The DMA will continue "looping" on the repeat portion until you stop the channel. Alternately, you can turn the audio-block-done interrupt off upon the second interrupt. In this case, you'll need to enable the interrupt each time before you restart the channel.

WELCOME INTERRUPTIONS

Some addresses in the Amiga are write-only, while others are read-only. You can move a value into a write-only address, but you cannot retrieve a value from it. The opposite is true of a read-only address. Some registers, such as the DMA, have separate addresses for reading and writing operations. Labeled `dmaconr`, the address to read the DMA is `$DFF002`. The write-only address is `$DFF096` and has the symbolic label of `dmaconw`. As you would say in your listing:

```
dmaconw equ $DFF096
dmaconr equ $DFF002
```

To start or stop audio channels, you must write to `dmaconw`. To start a channel, bits 9 and 15 of the DMA register must be set, as well as the enable bit of the audio channel you want to start. The bit numbers of the DMA register for the four audio channels are numbered 0–3, corresponding to the audio channel numbers. You can start or stop several channels simultaneously. For example, to start channels 1 and 2, you must move the value `$8206` to address `$DFF096` (`move.w #$8206,dmaconw`). To stop a channel, bit 15 of the DMA must be clear, as well as bit 9, but the enable bit of the channel that you wish to stop must be set. For example, you would use the instruction `move.w #$0001,dmaconw` to stop channel 0.

To determine if a channel is playing, read the status of the channel's enable bit at address `dmaconr`. For example, the sequence:

```
move.w dmaconr,d0
Btst.l #3,d0
bne.s ItsPlaying ;branch if channel is playing
```

will tell you whether channel 3 is playing. Never read from `dmaconw` or write to `dmaconr`, unless you want a guru error.

You should enable the audio-block-done interrupts as well. Each channel has its own interrupt-bit enable in the interrupt control register, `intena`. The interrupt control requester also has separate write (`$DFF09A`) and read (`$DFF01C`) addresses:

```
intena equ $DFF09A
intena_r equ $DFF01C
```

This register disables and enables the interrupts (allows or disallows them from occurring), but does not tell you the cur-

rent status of an interrupt. To check if an interrupt is occurring or to reset the interrupt (as you should do at the end of your handler routine), you must access the interrupt-request register, which has the following addresses for writing (`$DFF09C`) and reading (`$DFF01E`):

```
intreq equ $DFF09C
intreq_r equ $DFF01E
```

Improper set up and handling of an interrupt can be deadly. Always follow the proper three step procedure. First, install an interrupt handler for each bit in the `intena` register that you wish to enable. To do so, initialize an interrupt structure and pass its address and the `intena`-interrupt-bit number to the Exec's `SetIntVector()`. (The `intena` bits for audio channels 0–3 are 7–10, respectively.) Next, enable the proper channel's bit in `intena` by setting bit 15, and the bit (or bits if you are playing multiple channels) of the audio channel's audio-block-done interrupts that you want. For example, to enable interrupts for channels 0 and 1 use the command:

```
move.w #$8180,intena
```

To disable, clear bit 15 and set the channel's enable bit. The following command disables channel 3's audio block done interrupt:

```
move.w #$0400,intena
```

Finally, when an audio-block-done interrupt occurs on a channel, Exec calls its handler. The handler resets its interrupt bit via the `intreq` register so that future interrupts can occur. The bit assignments are the same as the `intena` register. For example, the handler for channel 2 would need to reset its bit as follows:

```
move.w #$0200,intreq
```

If you opt to fetch each word of data for a channel's data register yourself instead of using the DMA, the audio-block-done handler has an entirely different meaning. In this case, the audio channel sets its interrupt every time that it needs another word of data. The handler for a channel, therefore, should supply the next piece of data. Of course, the handler also must determine where to find the data and when to stop the channel. You still need to install the handler via `SetIntVector()`, enable the channel's bit in `intena`, and clear the bit in `intreq` at the end of the handler. The big difference is in how you start the channel. When you write the DMA register, set bit 15 and the bit of the channel to start, *but not bit 9*. Make sure that the channel's data register already has the first two bytes of the wave first sample as MSB. For example, to start channel 0 without DMA fetch:

```
lea WaveAddress,a0 ;the actual waveform data
move.b (a0)+,d0 ;the first sample point
lsl.w #8,d0
move.b (a0)+,d0 ;the next sample point ►
```



```

move.w d0,DATA_0 ;move it to channel 0's data register
move.l a0,WavePtr
;store the next sample's address for the handler
move.w #$8001,dmaconw ;start channel 0 w/o automatic data fetch
;When the audio block done interrupt occurs, the handler
;needs to get the next two bytes at WavePtr, put them in
;DATA_0, increment WavePtr by 2, and clear the
;interrupt for that channel. The handler continues doing
;this until it reaches the end of the waveform.
;It then turns off its audio channel.

```

Your final consideration is the audio device itself. If another program opens it, the audio device installs its own audio-block-done interrupt handlers, which replace yours. Therefore, you must force the audio device not to allow any tasks to use it. To do so, open the audio device *before* installing your handlers to force the device to install its handlers first. Next, lock all four channels with the audio device's CMD_LOCK command so that the audio device will turn away all future I/O requests. (If you cannot lock it, a task is already using the device.) Install your handlers, saving the return addresses from SetIntVector(). These returns will be the addresses of the audio device's handlers. When your program exits, reinstall the audio device's handlers using SetIntVector(). If the original address was 0, however, skip this step. Finally, unlock the four channels and close the audio device.

SOUND ADVICE IN ACTION

The example programs in the accompanying disk's Glatt directory demonstrate how to play what appears to be an 8SVX sample with a looping portion and an 8SVX sample with only a oneShot portion. I never really load the sample data, however, as it is part of the program (loaded into chip data when run). Because I chose to let the DMA feed the DAC and not to do double-buffering, I must accept the 28 KHz rate limit and the 131K sample-length limit. Therefore, neither the 8SVX oneShot nor the looping portion may be greater than 131K, however, the total of the two may.)

The examples have several audio routines that you can use verbatim in your own C or assembly applications. (Note that this example expects to run from the CLI for brevity. Nothing about the audio routines themselves, however, requires this.) The audio routines themselves (Audio.asm) are in 68000 assembly, but have C entry points so you can use them with C code. In fact, I included both C and assembly versions of the example. You must assemble the file Audio.asm, however, regardless of which you use. Now, let's step through the example.

The program entry point is at the label _main. I first open the DOS library and call SetupAudio() once to setup the audio hardware. In SetupAudio() at label S1, I get the base of the Exec library to access SetIntVector(). At label S2, I disable the four interrupts for the audio channels prior to installing my handlers; I do not want to be bothered by audio-block-done interrupts until I am ready. At label S3, I install the audio-block-done interrupt for audio channel 0. Note that d0 equals 7; 7 is the bit in the intena register that pertains to channel 0. Look at the interrupt structure, int0. Its IS_CODE field points to the function, audioInt. When the audio-block-done interrupt for channel 0 is set, the program calls audioInt(). Note that the IS_DATA field points to my own structure called a LoopStruct that both PlayAudio() and audioInt() use. Each channel has its own LoopStruct. I set the other channels up at labels S4, S5, and S6. At S7, I set a

flag to indicate that the program is about to save the original handlers that it may have replaced. If this bit is already set, then I must have already called SetupAudio() without calling FreeAudio(). This flag bit makes let's me call SetupAudio() and FreeAudio() safely and repeatedly, at any time.

I am now ready to play back audio samples. The sample data is at the labels Wave1shot, Wave1loop, and Wave2shot. The oneShot portion of the first 8SVX sample, Wave1shot is 512 bytes. Wave1loop is the Repeat section of the sample and is 256 bytes long. The sampling frequency is 18000 Hz. Wave2shot will be analogous to an 8SVX sample with a oneShot portion only.

To convert the sampling rate (in hertz) to the wave's period I use two routines. RateToPeriod() simply converts the rate to a period in nanoseconds, so the program can use long multiplies (no floating point) when calculating periods without losing too much resolution. At label M4 in _main, I pass the rate to RateToPeriod() and save it in d7. Now, when I play this wave, I can transpose it anywhere from an octave up (+12 half steps) to an octave down (-12 half steps) using the function TransposePeriod(). Note that a step value of 0 returns a period that plays the wave at its original pitch. At label M5, I play the wave at its original pitch on channel 0.

In PlayNote(), at P1, I retrieve the custom chips' addresses so that I can reference the intena, intreq, and dmaconw registers. At P2, I find the current channel's LoopStruct (size = 16 bytes), and from that structure, I determine this channel's pointer audio register address. At P4, I turn the channel volume down in case the channel is playing another wave. To simply stop the channel might cause a clicking noise as the audio amp closes down too quickly. (You can hear this when you use the audio device.) Note that I lower the volume by half and then turn it off to minimize the click.

At P5, I stop the channel by writing its mask with bit 15 clear to the dmaconw register. Now, I can safely alter the location and length registers. After all, when the channel is stopped, the DMA will not access the registers and audioInt() will not be executed. Note that if the location is 0, then I exit PlayNote, effectively stopping the channel without starting a new wave. (This is where StopChan() exits PlayNote().) Also, note how at P7, I divide the length of the sample in bytes by two because the audio hardware needs to know the number of words not bytes.

At P10, I setup the channel's LoopStruct. When the channel's block done interrupt occurs, audioInt will check LoopStruct to determine whether to play a repeat portion or stop the channel after the oneShot portion. At P11, I set the DMA bit and the start bit of the mask in d0, so when I write this to the dmaconw register, the channel will start playing. (First, however, I clear any interrupt that might have occurred prior to my "stealing" this channel by writing my mask with bit 15 clear to intreq.) Then at P13, I set bit #15 of the mask and write intena. Note that this enables the block-done interrupt for the channel, but does not execute it. (Writing the same value to the intreq register subsequently would cause an immediate block done interrupt.)

Finally, at P14, I start the channel. The DMA grabs the values that I placed in the hardware registers, puts them into its backup registers, and causes a block-done interrupt (Exec calls audioInt() from a 68000 interrupt.)

The task halts while that interrupt is serviced. Let's take a look at what audioInt() is doing. Because this is an interrupt

Continued on p. 46

Recycle Your Sprites

Put more action on screen faster by reusing your sprite channels vertically.

By Leo L. Schwab

A SHORT TRIP through *The Amiga Hardware Reference Manual* (Addison-Wesley) will convince you that the Amiga has a lot of hidden tricks. Many of these, however, stay hidden because they are either poorly understood or not supported by the OS.

One of these well-kept secrets is that you can reuse sprites vertically. The Amiga hardware can use a sprite channel for the entire height of the display region. Further, this use need not be continuous; the hardware can use the same sprite channel to make several images appear at various locations on the screen. The OS-supported method of reusing sprites is through VSprites, which are part of the GELS system. Unfortunately, VSprites are cumbersome and somewhat slow. They work as fast as they want, not as fast as you would like. Investigating the format of sprite data, I hit upon another approach to sprite reuse that is friendly to the system, easy to implement, and of reasonably high performance.

MINCING WORDS

Sprite data is laid out in chip memory as shown in Figures 1 and 2. The first two words of "image" data are position and control information. As you can see, the format of these words is identical to the hardware registers SPRxPOS and SPRxCTL:

Bits	Function
SPRxPOS (image word 0):	
15-8	Bits 7-0 of the vertical start (VSTART) position
7-0	Bits 8-1 of the horizontal start (HSTART) position
SPRxCTL (image word 1):	
15-8	Bits 7-0 of the vertical stop (VSTOP) position
7	SPRITE_ATTACHED bit
6-3	Unused (set to 0)
2	Bit 8 of the vertical start (VSTART) position
1	Bit 8 of the vertical stop (VSTOP) position
0	Bit 0 of the horizontal start (HSTART) position

Immediately following this information are the actual words of image data, which are treated in pairs. The first word in an image pair is the low-order word (bitplane 0, so to speak), and the second word is the high-order word. A

In-Memory Sprite Data Format

```

UWORD spritedata[] = {
    0x0000, 0x0000,  // First set of control words
    0x0550, 0x0FF0,  // Image data for first sprite
    0x1EE8, 0x1FF8,
    :
    0x0000, 0x0000,  // Next set of control words
    0x0550, 0x0FF0,  // Image data for next sprite
    0xF879, 0xFF23,
    :
    0x0000, 0x0000  // Terminating control words
};
  
```

This data must be in CHIP RAM.

Figure 1. The arrangement of sprite data in chip RAM.

Sprite Register / Control Word Format

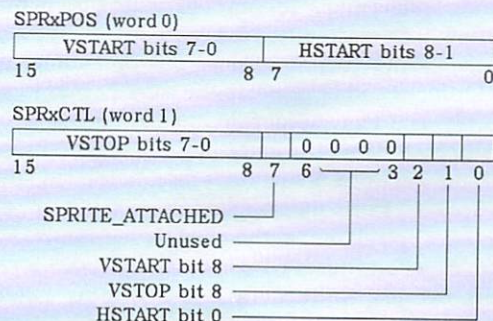


Figure 2. The format of position and control words.

pair of words forms one line of a sprite.

Here's where it gets interesting. Following the image data are two more words. Ordinarily in the case of SimpleSprites, these two words contain zeros, which indicate the end of the sprite, and are the last two words of sprite data. These two words are, in fact, position and control words for the next use of the sprite channel. In format, they are identical to the position and control words at the beginning of the sprite data. It is perfectly legal to place new position and control

information here, followed by more image data, then more control words and more imagery, and so on.

Usually, you manipulate the sprite data control words via the graphics.library calls `MoveSprite()` and `ChangeSprite()`. Internally, `MoveSprite()` calls `ChangeSprite()`. `ChangeSprite()` computes the position and control words, writes them to the sprite data, and resets the appropriate hardware registers to point to the top of the sprite data. At first glance, you might think that you could simply reset the `posctldata` field in the `SimpleSprite` structure to point to the successive control words and call `MoveSprite()` to manipulate them. Recall, however, that `MoveSprite()` calls `ChangeSprite()`, which resets the hardware's idea of the top of the sprite data. Resetting `posctldata` and calling `MoveSprite()` could leave the hardware pointing somewhere other than the top of your sprite data.

If you are cautious about the sequence in which you make the calls, taking care to call `MoveSprite()` for the "top" sprite last, then you can use `MoveSprite()`, although doing this kind of bookkeeping is a bit cumbersome. The ideal function would rewrite the control words in the sprite data and not touch the hardware.

GET MOVING

The example program (see `ms.c` in the accompanying disk's Schwab directory) contains a function, `MyMoveSprite()`, that fits the bill. Semantically identical to `MoveSprite()`, it computes the correct values for the control words and writes them to the address contained in the `SimpleSprite`'s `posctldata` field. `MyMoveSprite()` does not touch the hardware. Its method for computing the control words is rather interesting. At the hardware level, sprite coordinates are specified in low-resolution, noninterlace pixel units. The origin of this coordinate system is a point off the physical display in the upper-left corner. You can bring sprites into a known coordinate system by adding the offsets of the currently active View to the supplied `x,y` coordinates. `MyMoveSprite()` does this by adding the View fields `DxOffset` and `DyOffset` to the passed `x` and `y` values, respectively. The View offset fields always represent low-resolution, noninterlace pixel units.

If a pointer to a `Viewport` is also supplied, then `MyMoveSprite()` treats the supplied coordinates not as lo-res, noninterlace pixels, but as the pixel size currently prevailing in that `Viewport` (this is consistent with `MoveSprite()`'s behavior). Because sprites only relate to lo-res noninterlace pixels, however, a program must convert the `Viewport` coordinates to the closest possible sprite coordinates. `MyMoveSprite()` accomplishes this by examining the `Modes` field in the `Viewport` structure and scaling the coordinates accordingly. If the `Viewport` is high-resolution, the routine divides the `x` value by two. If the `Viewport` is interlaced, the `y` value is divided by two. (Be warned: This approach may not work under Kickstart 2.0. I have not investigated it yet.)

The offsets of the `Viewport` relative to the View must be added to the `x,y` coordinates, as well. In `MyMoveSprite()`, I add the `DxOffset` and `DyOffset` fields in the `Viewport` structure to the `x` and `y` values, respectively. These offsets are also expressed in the pixel resolution prevailing in the given `Viewport` and must be converted to sprite coordinates.

Once the routine computes the final sprite coordinates, it creates the position and control words in a straightforward, if slightly clumsy, operation. The position word (word 0) contains bits 7-0 of the `y` value (`VSTART`), and bits 8-1 of the `x`

value (`HSTART`). Therefore:

```
pos = (VSTART << 8) + (HSTART >> 1);
```

(assuming `HSTART` is in the range 0-511.)

The control word (word 1) is more complicated. The high eight bits, which are bits 7-0 of the `VSTOP` value, are easy:

```
ctrl = (VSTART + spriteheight) << 8;
```

Three bits require specific locations. Bit eight of `VSTOP` must be written to bit one of the control word:

```
if ((VSTART + spriteheight) & 0x100)
```

```
    ctrl |= 2;
```

Bit eight of `VSTART` belongs in bit two of the control word:

```
if (VSTART & 0x100)
```

```
    ctrl |= 4;
```

Bit zero of `HSTART` must reside in the control word's bit zero:

```
ctrl |= HSTART & 1;
```

Finally, you should preserve the `SPRITE_ATTACHED` bit. This is an OS-supported trick up the Amiga's sleeve that permits the use of 16-color sprites. While creating and using attached sprites is beyond the scope of this article, `MyMoveSprite()` preserves the `SPRITE_ATTACHED` bit.

```
ctrl |= spriteptr ->posctldata[1] & SPRITE_ATTACHED;
```

(Be careful: Unlike `MoveSprite()`, `MyMoveSprite()` does not properly support the `SPRITE_ATTACHED` feature. If you use `MyMoveSprite()` with attached sprites, you will need to call `MyMoveSprite()` for both sprites in the attached pair.)

Having created the position and control words, `MyMoveSprite()` writes them to the sprite data:

```
spriteptr ->posctldata[0] = pos;
```

```
spriteptr ->posctldata[1] = ctrl;
```

The sprite hardware will pick up this change when the sprite DMA channel reads the new control words, and the hardware moves the sprite to the new location.

GO WITH THE FLOW

The theory sounds great, but how do you put it to practical use? Set up your sprite image data in chip RAM starting with a pair of control words, followed by the first image, then two more control words, then another image, and so on. After the last image, place two final control words containing the value zero to tell the sprite DMA hardware to stop fetching sprite data. Next, create a `SimpleSprite` structure for each image you embedded in the sprite data. For each `SimpleSprite` structure, you initialize the `posctldata` field with the addresses of each pair of control words. This way, you have a `SimpleSprite` structure managing each sprite image embedded in your sprite data.

Now, allocate a sprite channel with the graphics call `GetSprite()`:

```
if (GetSprite (&topsprite, -1) < 0)
```

```
    error (NO_SPRITE);
```

Use the graphics call `ChangeSprite()` to point the system at the top of your sprite data and tell the hardware to start processing it:

```
ChangeSprite (vport, &topsprite, spritedatatop);
```


At this point, at the very least, the first sprite in your sprite data will be visible. If you preset all the successive control words as well, the rest of your images will appear, too. Finally, you can call `MyMoveSprite()` on any one of the `SimpleSprite` structures, and the corresponding sprite image will move to the desired location.

Alternatively, you can initialize the positions of all the sprite images by calling `MyMoveSprite()` for each of them before calling `ChangeSprite()`. When you finally call `ChangeSprite()`, all the sprite images will immediately appear in their specified locations.

If you want to be sure all the sprites move simultaneously, you can double-buffer your sprite data. To do so, you create and maintain two identical sets of sprite data, then call `ChangeSprite()` to display one set and `MyMoveSprite()` to modify the other. When your modifications are complete, you call `ChangeSprite()` to display the newly modified set, and start changing the old set with `MyMoveSprite()`... and repeat.

TRIP UPS

The most obvious limitation with reusing a sprite channel is that you must not supply `MyMoveSprite()` with coordinates that will cause the imagery to collide vertically. The highest position a sprite image can occupy, therefore, is one line below the last displayed line of the previous sprite image on the same channel. Remember: The sprite DMA hardware moves through your sprite data sequentially as the video beam moves down the display. Because you cannot make the

video beam back up, you can only specify locations on successively lower display positions. While you have complete freedom in horizontal positioning, subsequent images in the sprite data must appear in successively lower positions on the screen. Because of the hardware set up, successive sprite imagery must be vertically separated on the screen by at least one blank line.

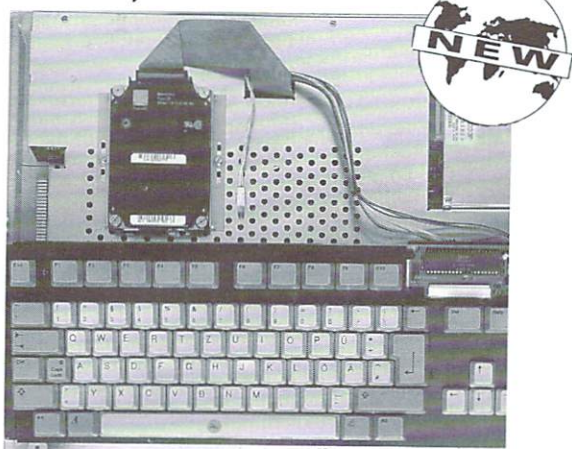
The function `MyMoveSprite()` modifies the control data directly. This modification can theoretically happen while the hardware is reading the data. This may result in the sprite imagery being corrupted or disappearing for one video field. You can avoid this problem by using video beam avoidance techniques or by double-buffering your sprite data.

Despite these limitations, this is a useful trick that can help make certain graphic effects easier. For example, you can use this feature to create a multiplane side-scrolling star field using nothing but sprites. Thus, you can have a very effective scrolling background of stars, and still have the whole standard display available unhindered. (European demo programs do this all the time.) Use it to add that little extra embellishment that lets your program, and the Amiga, stand out just a bit more. ■

Leo L. Schwab is the principal programmer behind Disney Presents... The Animation Studio and creator of many PD screen hacks. He can frequently be seen at computer shows wearing a cape and terrorizing IBM reps. Contact him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458 or on BIX, People-Link, or USENET as ewhac.

ARRIBA HD 20

internal for the AMIGA 500!



Interface:

6-layers Multilayer
SMD-technology
Auto config
Autoboot capability with Kickstart 1.3
Compatibility with internal + external expansions

20 MB-Harddisk:

16 Bit-AT-Task-File-Interface
23 ms average seek-time
No external power supply

Pre'spect Technics Inc.

P.O. Box 670, Station H
Montreal, Quebec H3G 2M6
Phone: (514) 954-1483
Fax: (514) 876-2869

714-283-0498
800-942-9505

AAMIGA WAREHOUSE

714-283-0499
800-942-9505

15448 FELDSPAR DR., CHINO HILLS, CA. 91709

"We believe there is a cure for MSDOS"



MASTER 3A-1 Disk Drive

\$79.95



GOLDENIMAGE

HAND SCANNER
RC500 (A501 clone)
OPTICAL MOUSE
OPTO-MECHANICAL

MASTER 3A-1D
2-8 MB BOARD (A2000)

WE WILL BEAT ANY PRICE ON ANY OF THESE PRODUCTS.

WE WILL BEAT ANY ADVERTISED PRICE!

AND JUST ABOUT ALL UNADVERTISED PRICES ALSO.

MEMORY UPGRADES

DRAMS

64x4 - 120/100/80/70
256x1 - 120/100/80/70
256x4 - 100/80/70
265x4 - 100/80 Page Zip
1M x 1 - 100/80/70

SIMMS

1x8 - 12/10/80/70
4x8 - 80/70
GVP SIMMS TOO!

A3000 STATIC ZIPS

1x4-80/70.....\$42.95
256x4-80.....\$6.95

"FOR SOFTWARE GO TO THE REST.
FOR HARDWARE CALL THE BEST!!!"

Why purchase from a large company where YOU are just a number? Buy your AMIGA hardware from guy's that own AMIGA's and know how to use them.

INTERNATIONAL ORDERS
SAME DAY SHIPPING
UPS - RED, BLUE, GROUND
C.O.D. ACCEPTED ALONG WITH



1-800-942-9505



GRAPHICS HANDLER

The Deep-Bitmap IFF Standard

By Perry Kivolowitz

FOR ALL OF its generality, the original IFF ILBM specification was closely tied to the specifics of the Amiga's native display capabilities and did not accommodate the added data requirements of true color image processing. To keep pace with the times, ASDG has (in cooperation with others, most notably Carolyn Scheppner and Justin McCormick) developed and established a standard for deep bitmaps, as well as for images with 6 to 8 bitplanes as a special case.

While some of this new code has been adopted by Commodore as a standard, much is convention that ASDG has developed. By presenting it publicly, I hope to garner support for these ASDG conventions so that they may join the endorsed standard. They are not a complete solution, however; the current deep-bitmap standard is just barely adequate. As you will see, work towards standardization must still be done in some areas (such as Alpha channel support), and in others, assumptions were made that will limit the extendability of the standard.

IFF EXCAVATIONS

The original IFF ILBM (InterLeaved BitMap) specification provided for at most eight bitplanes of image data. (For a thorough discussion, see the *Amiga ROM Kernel Manual: Includes and Autodocs* from Addison-Wesley.) The format is called interleaved because the data is stored by interleaving a single scanline's worth of data from plane 0 followed by a single scanline's worth of plane 1, and so on. From the top of the image (beginning of the file) to the end of the image (end of the file), the data comprising each scanline's planes is interleaved across the number of lines in the image.

No one plane has color significance attached to it. Rather, one bit in each plane contributes to an index into a color map (CMAP) that, in turn, tells you which color should be displayed at a given pixel. Images written in the original IFF specification, therefore, are mapped. The maximum number of significant entries in a color map is defined by $2^{\text{bitmapdepth}}$. For example, a 3-bitplane-deep image should have at most eight (2^3) entries in its CMAP chunk. For some new software and hardware, however, the traditional CMAP chunk does not make sense. For example, a CMAP for a 24-bit color image would be 50,331,648 bytes long, an unweildy and ridiculous size.

(CMAPs are notorious trouble spots for programmers. Remember that each CMAP entry is three bytes long, and all IFF chunks begin on word boundaries. If a CMAP length comes out odd, you should write its chunk length as odd and append a single padding byte after the entire chunk.)

The altered IFF standard allows for deep bitmaps with 12 or

more bitplanes where the actual number of bitplanes must be a multiple of three. Under this rule, 12-, 15-, 18-, 21-, and 24-bitplane formats are completely acceptable deep-bitmap IFF formats. The standard begins at 12 bitplanes because it fits naturally with the Amiga's 12-bit palette width and because no developers voiced a desire to begin at fewer than 12 bitplanes.

The upper limit, 24 bitplanes, is a practical limit. You can exceed it, but no current devices take advantage of the information. At ASDG, for example, the IFF readers will correctly process an IFF file deeper than 24 bitplanes. This code simply discards (as yours should) the *least* significant bitplanes beyond 24.

The same applies if you want your application to process fewer bitplanes than are supplied: Discard the least significant planes beyond the depth you are willing to accommodate. Note that you will still have to correctly interpret any Color Look-Up Table (CLUT) chunks which may be present (more on CLUTs later). Do not get too comfortable with this method; it is the least appealing for color reduction.

According to Commodore mandate, the file's bits must be ordered as an interleaved bitmap from the least significant bit to the most significant bit, for each of the red, green, and blue values. For example, for a 12-bitplane image, you would specify planes 0 through 3 of red data for scanline 0, followed by planes 0 through 3 of the same scanline's green and blue data, respectively. You repeat the same order for each successive scanline until you specify the entire image. Note that you must specify to which color each plane in the data corresponds, but that you need not when describing a bitmap in the original IFF standard. In a 12-bitplane image, the first four planes per scanline correspond to the red information in the image. Therefore, without using a CMAP you can assemble the image and have it appear reasonably correct. This is an example of noncolor-mapped image data.

The first limitation in the current deep-bitmap standard is that it cannot support formats in which the component colors do *not* have the same resolution. All bitmap depths at 12 or beyond must be a multiple of three with each color component using exactly one-third of the image data. This is unfortunate because some color models lend themselves to nonequal color distributions. This is not a major restriction, however, because IFF was defined as working within the RGB color model. Clearly, if you need nonequal RGB components, making the bitmap depth triple the deepest desired component depth and padding any unused planes will sidestep this limitation. Some alternative image file formats, such as TIFF, allow you to specify the individual depth of each color component directly. I and my ASDG coworkers suggest

that a similar capability be implemented in IFF.

A GLUT OF CLUTS

To determine the color of an individual pixel in the original specification, you had to assemble the component planes for a given pixel into an index number, then read out the pixel's actual color from the color map table entry specified by the index number. The ability to perform some type of color indirection is extremely important to many applications even when dealing with such noncolor-mapped images as deep bitmaps. The CLUT chunk provides this color indirection capability. CLUT chunks have the fixed structure shown below:

```
struct=CLUT {
    ULONG Type;
    ULONG Reserved;
    UBYTE LUT[256];
};
```

The fact that this structure *is* fixed is the second limitation in the deep-bitmap standard. Specifically, the CLUT chunk's fixed structure limits CLUT resolution to eight bits and the bitmap depth for which CLUTs are usable to 24 bitplanes.

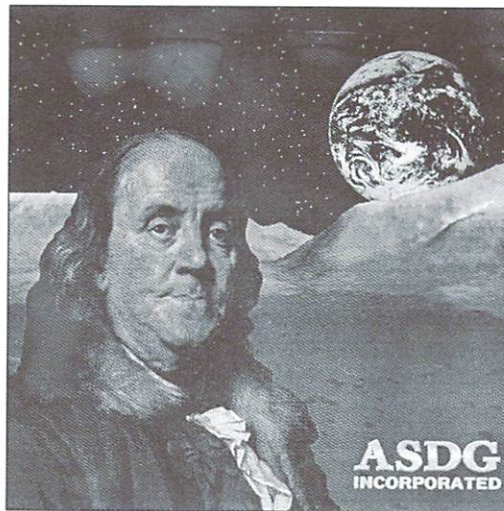
The following six CLUT types are currently defined:

```
#define CLUT_GREEN = 0
#define CLUT_MONO > 0
#define CLUT_RED > 1
#define CLUT_GREEN > 2
#define CLUT_BLUE > 3
#define CLUT_HUE > 4
#define CLUT_SAT > 5
```

Of these, ASDG (and all developers I know) never uses the hue and saturation chunks. ASDG uses the monochrome CLUT to filter gray scale image data and the red, green, and blue CLUTs to filter the corresponding type of color data. The effect of the color balancing controls in ASDG's Art Department programs were implemented using CLUTs. I prefer this to modifying the data to reflect, for example, an increase in brightness, because you can easily reverse modifications to a CLUT but not to the raw image data.

When your program finds a CLUT, it uses the CLUT as a filter for a specific segment of the image data. For example, the red CLUT color maps only the red color data.

As indicated above, CLUTs always have 256 entries even if the bitmap to be stored has fewer than 256 shades (8 bitplanes) of each primary color. In such cases, take care to fill the CLUT uniformly, using the entire range of indices (from 0 to 255). The entire CLUT must be valid through indices 0 to 255. As a result,



An eight-bit gray-scale (deep-bitmap) image.

your program may have to do some interpolation.

When interpreting raw image data that is less than eight bits wide per primary color, you must expand the raw data to the range of 0 to 255 to determine which CLUT entry to use. For best results, use a method that allows both a pure dark (0) and a pure light (255). Simply shifting left by a number of bits does not accomplish this. ASDG uses the algorithm:

Let D be a red, green, blue, or gray component of a pixel. D is N bits wide where $4 \leq N \leq 8$. Then $\text{Index} = (D << (8 - N)) \mid (D >> (2N - 8))$

For example, a red value of 0 from a 12 bitplane-file becomes index 0 into the red CLUT (if present), but a value of 15 becomes 255, giving you the pure light and dark you need. Simply shifting a value of 15 left by 4 bits gives you 240, not 255.

In some rare cases you may want to use both a CLUT and a CMAP. For example, you could store a four-bitplane gray-scale image with both a MONO CLUT and a CMAP. The CMAP might map the 16 possible color registers to gray scales. Having a CMAP present as well as a CLUT allows any standard Amiga viewer program to read the image. If both a CLUT and a CMAP are present, the program should always give precedence to the CLUT. CLUTs are optional, but if present, they should be interpreted. If they are not present, your program should assume a linear or neutral setting.

The hue and saturation CLUTs can interact with the individual color CLUTs. No order for processing CLUTs (other than order of appearance) has been defined, leaving room for alternative methods of interpretation and, as a result, incompatible files. While this is not currently a pressing problem (no one I know of uses the hue and saturation CLUTs), a definite processing order should be discussed and defined.

In an IFF file, the chunk length of a CLUT chunk should always be 264 bytes. While not yet endorsed by Commodore, the extended CLUT structure

```
struct=CLUT {
    ULONG Type;
    UWORD NEntries;
    UBYTE NBits;
    UBYTE NBytes;
    UBYTE LUT[];
};
```

is one with which we can grow. The CLUT would have the ►

number of entries specified in the NEntries field. Each entry would be NBytes wide, of which the NBits least significant bits would be counted. This proposed structure has the benefit of being backwards compatible with the older CLUT definition when the CLUT width is one byte. New CLUT types, such as those below, can be defined to differentiate a variable-width CLUT from the original fixed-width CLUT:

```
#define CLUT_VARIABLE_WIDTH_GREEN= 16
#define CLUT_VARIABLE_WIDTH_MONO> 16
#define CLUT_VARIABLE_WIDTH_RED> 17
#define CLUT_VARIABLE_WIDTH_GREEN> 18
#define CLUT_VARIABLE_WIDTH_BLUE> 19
#define CLUT_VARIABLE_WIDTH_HUE> 20
#define CLUT_VARIABLE_WIDTH_SAT> 21
```

BODY AND BMHD ISSUES

Apart from the bit ordering, writing the BODY chunk of a deep-bitmap IFF file is the same as writing any other IFF BODY. Remember, however, that every plane and scan line *must* begin on a word boundary. As the original IFF specification says, "The fields w and h (of the bitmap header) indicate the size of the image rectangle in pixels. Each row of the image is stored in an integral number of 16-bit words."

The intended use of the bitmap header's pageWidth and pageHeight fields, specifying a suggested display size, does not make the transition into handling large nondisplayable bitmaps. ASDG always sets these to the very safe 320×200 size. With deep bitmaps, w and h specify the size of the image, not pageWidth or pageHeight. ASDG does not use the x and y fields, which specify the image's position in relation to a larger enclosing image, setting them to 0.

The xAspect and yAspect fields can be helpful in determining if the image is intended to be displayed in an interlaced format. For Amiga-displayable images, however, desired screen formatting information should come from a CAMG chunk. In general, you should set xAspect and yAspect as accurately as you can so reader programs that monitor these fields receive valid data. Note that programs that do not pay attention to these fields should be prepared to handle nonfactored aspect ratios. For example, a reader program should be able to handle an aspect of 20 to 22 or 100 to 101, as well as the prime-factored 10 to 11. Make sure your image reader, however, can handle compression types other than 0 and 1 gracefully. If your program cannot understand a compression type, it should report this to the user rather than load the file and assume a known type of compression.

CAMG AND ID CHUNKS

Unlike for standard images, you do not need a CAMG chunk for a deep-bitmap image. In fact, the absence of a CAMG chunk may be the only way you can tell the type of data contained in the given file. For example, the only way to tell that a 6-bitplane file is a color-mapped 64-color (distinct) image is by noting that CAMG chunk is either absent or specifically does *not* say the image is HAM or Extra_Halfbrite. By the time your program reads the BODY chunk it should know exactly what type of image it is reading and how to interpret the image.

Some IFF writers out there, such as DeluxePaint III (Electronic Arts) and several public-domain programs, create confusing CMAP chunks when writing Extra_Halfbrite (EHB) and HAM images. When we save an EHB image, we save

only 32 color registers because the second set of 32 are defined by the first. If you wish to write all 64 color registers, make sure you write the correct values in the latter half. DeluxePaint III, for example, places random numbers in the second 32 registers which confuses nonAmiga IFF readers if they do not notice the CAMG EHB flag. When saving HAM files, we save only 16 colors registers because this is all that may be used. (Some PD programs save 64 color registers for no apparent reason.) Remember, if you are saving a HAM or EHB image, make sure you include the appropriate CAMG chunk so that 6-bitplane HAM and EHB files can be distinguished from 64-color color-mapped files.

Two IFF chunks (ANNO and AUTH) can be used to insert comments into an IFF image file. ASDG strongly recommends that you use the ANNO chunk to insert a string that lists the identity of the last program that wrote this file. You can make the AUTH chunk available to the user for other uses such as the artist's name. When writing an IFF file, we strongly recommend that you never carry forward another program's ANNO chunk. Doing so would make keeping track of the last writer impossible. Being able to determine which program wrote the image will help if your program finds file incompatibilities.

THE NEWTEK CONNECTION

NewTek's Digi-View 3.0 software stores 21-bitplane raw image data in a form completely compatible with the deep-bitmap standard. Unfortunately, it writes the bits in the opposite order from Commodore's standard. Even worse, you cannot determine if you are processing one of these files until the erroneous results appear on the screen. Thankfully, Digi-View 4.0 and the Video Toaster's software write their bits in the Commodore-endorsed order and should be compatible with the extended deep bitmap specification.

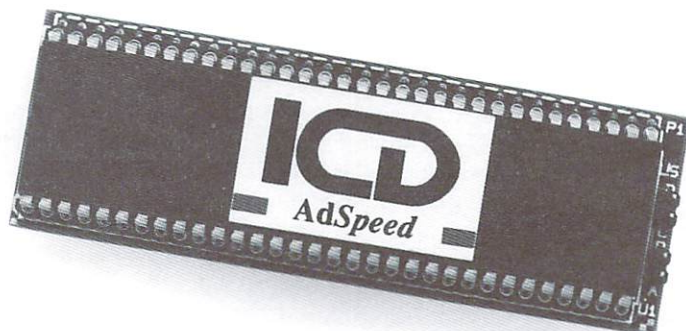
The differences are only semantic when it comes to two newer variations on the IFF format supported by ASDG and NewTek (but not endorsed by Commodore). While named differently (AHAM and ARES for ASDG; Dynamic HAM and Dynamic Hi-Res for NewTek), the file formats are identical. ARES (Dynamic Hi-Res) stores the image as a standard high-resolution image (bit set in CAMG) with four bitplanes. The format uses a standard CMAP chunk, plus a new chunk called the CTBL prior to the BODY. The CTBL contains one 16-color palette (32 bytes, each color occupying a 16-bit word with the color information defined in the form of a 12-bit RGB triple) for each scanline in the image. AHAM (Dynamic HAM) stores the image as a standard HAM image (bit set in the CAMG) with 6 bitplanes. Again it contains a standard CMAP chunk and a CTBL chunk prior to the BODY. To summarize these modes: If you find a CTBL chunk, it means you are processing a dynamic mode file. Check the CAMG to figure out which type of dynamic file you are working on.

Although it is not required, we strongly recommend that you place the darkest color of each palette in the color register 0 position. This provides for a less distracting side border when displaying nonoverscanned "dynamic" images.

Should you need more information on the emerging deep-bitmap IFF standard or wish to offer suggestions, speak up in the amiga.dev/iff conference on BIX. I'll be waiting for you. ■

Perry Kivolowitz is co-founder of ASDG Inc., which specializes in color image processing. Contact him at The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458 or as perry on BIX.

This is the most cost effective way to
increase the speed of your computer.
AdSpeed™!



AdSpeed

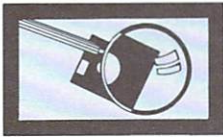
ICD expands its line of innovative enhancement products for the Amiga® with the introduction of **AdSpeed**, a low cost, full featured 14.3 megahertz accelerator for all 68000-based Amiga computers.

AdSpeed differs from other accelerators by using an intelligent 16K static RAM cache to allow zero wait state execution of many operations at twice the regular speed. All programs will show improvement. No 68000 or 68020 accelerator without on board RAM will make an Amiga run faster.

AdSpeed continues ICD's tradition of providing the best product available. These are some of the features that set it apart from the rest:

- Works with all 68000-based Amiga computers, including the 500, 1000, and 2000.
- Simple no solder installation — just remove the computer's 68000 and plug **AdSpeed** into its socket.
- Low power, high speed CMOS 68000 CPU for full 100% instruction set compatibility.
- Software selectable speeds, with a true 7.16 megahertz mode for 100% compatibility. Switches speeds on the fly without rebooting the computer.
- 32 kilobytes of high speed static RAM — 16K of data/instruction cache and 16K of cache tag memory.
- Full read and write-through cache for greatest speed.
- Bus monitoring to prevent DMA conflicts.
- ICD's famous quality, dependability, and support.
- Worlds smallest 68000 accelerator. (Photo above is actual size).

ICD, Incorporated 1220 Rock Street Rockford, IL 61101 USA
(815) 968-2228 Information (800) 373-7700 Orders (815) 968-6888 FAX
AdSpeed is a trademark of ICD, Inc. Amiga is a registered trademark of Commodore-Amiga, Inc.



REVIEWS

CygnusEd Professional Release 2.11

Flexible and function filled.

By Tim Grantham

PREVIOUS VERSIONS OF the CygnusEd Professional (CED) text editor deservedly had a loyal following of Amiga programmers. Release 2.11 brings them good news: It adds user-definable colors, a high-speed global search-and-replace, support for alternate fonts, and a wonderful Undo/Redo capability. It also significantly enhances many features of previous versions, including macros, file security, and AmigaDOS/ARexx support.

Potential new users, however, may not be able to overlook the limited display choices offered by the program. CED can display up to only ten views at a time of the files (or variations of one file) being edited. Ten views may sound like a lot, but it's a ceiling I've bumped my head on several times. Even more annoying, CED forces all views into one screen or window. You can resize the views vertically down to one line of text and a title bar, but they eat up a fair amount of screen real estate. I prefer text editors that let me view a full screen of text for each file in memory, such as Rick Stiles's shareware editor UEdit. CED does have an auto-expand feature that automatically resizes the inactive views to provide the maximum possible size for the active view, but I find the clutter of inactive views an irritation, even in hi-res.

DESIGN YOUR OWN

CED's limitations in display lie in sharp contrast to the rest of the program, which is a model of flexibility. To customize it, you can create, load, and save macros, then bind them to any key. You can also create multiple preference files that contain settings for screen size, display colors, font, and so on, plus edit modes, save mode, task priority, macro definitions, tabs, and many other variables. A2024 monitor owners can increase the

screen size up to 1000 by 800, as well.

You can also add functions to CED using its extensive ARexx command set. For example, I wrote an ARexx script that positions my CED text cursor at the location of each error in my Aztec C source code and allows me to correct the error before moving on the next one. Once I'm satisfied with the corrections, I can start the cycle again. Plus, the included utility CB2RX lets you add AmigaDOS clipboard support to any program that speaks ARexx.

CED makes life easier for programmers in many other ways. It has a find-matching-bracket function, columnar block cut and paste, backward searching abilities, user-selectable text scrolling speed, selectable tabs (fixed or custom width) versus spaces, jump to line (an essential command for my ARexx), split views, and the ability to edit binary files. If you often type commands in the wrong case, you'll appreciate that CED lets you toggle the word at the cursor between upper- and lowercase.

CED Release 2.11 adds new file, print, and font requesters. One of the best I've used, the file requester has several good features: the ability to hide or show files in the listing using both AmigaDOS and Unix wildcard characters, a full listing of all physical and logical devices, the ability to select multiple files, the display of file sizes, and the use of keyboard equivalents. On the negative side, CED doesn't highlight selected files and does cache the listing. The latter provides fast display but doesn't keep up with changes to the directory's contents. You can force a reread of the directory with a gadget for that purpose. As a nice bonus, ASDG includes full documentation for the freely distributable runtime library used to create the file requester. Unfortunately, the print requester is almost identical to the file requester, which can cause confusion.

Macros in CED 2.11 can now collect data entered from requesters and can, like AmigaDOS and ARexx commands, be bound to any key. I created one macro to get around a CED limitation:

The macro combines two commands, Open New and Open, so that I can load a new file into a new view with one keystroke. A helpful addition would be a way to get a listing, by key, of all macros from within CED.

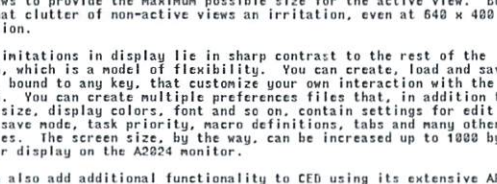
The Undo/Redo command is terrific. It lets me unwind almost all changes to a file in memory, right back to the beginning if necessary, which I've always wanted to be able to do. The catch here is memory. According to ASDG, each operation on the text, even entering a single character, consumes about 50 bytes from the undo memory pool. Multiply that by the number of files loaded into CED and you begin to see the RAM required. But if you've got an elephant's memory in your machine, go for it.

Getting files in and out of the program is easy. When run from the CLI, CED attempts to load an environment (preferences) file with the same extension as the file to be edited. If, for example, I run CED with a file called example.ts as an argument, CED will look for an environment file called s:ceddefaults.ts, configure itself with that file, and then load example.ts. This enables you to configure CED to work like the editor used to create the loaded file automatically. If you prefer, you can restart CED with a hot-key combination, providing you shut the program down appropriately. So that you don't forget to save your files, Autosave lets you set a time-lapse reminder. If you save files with icons, you can set the icon tool from CED. Should your system crash before you save, RecoverCEDFiles, a supplied utility, searches through memory for CED text files orphaned by the crash. I only wish the program's installation script let me specify where I want to put the CED executable and its associated files.

CED supports limited printing capabilities, as is true of most text editors. You can set typetypes using function keys, but cannot display them. Tabs can be converted to white spaces and vice versa. Print function output can be sent

SNAKES IN THE GARDEN

CED features extensive ARexx support, with almost every menu selection callable from an ARexx script, plus other commands that aren't in the menus, such as Jump To Column. I found the ARexx implementation strategy a little odd. CED uses the exact names of the menu items as the names of the ARexx-callable commands. While this might make most scripts easier to read, it means that you cannot directly call menu items that change their texts during operation. For those items you must resort to a "menu n n n" call, where the arguments to menu specify the menu, menu item, and menu sub-item, respectively. In a similarly odd vein is the "status n" call, in which n is a number that varies according to which piece of status data is required. Why not simply have a call to status return a structure describing the complete CED environment? Most annoying, CED currently provides only one ARexx port name: `rex.xced`. This causes a problem if more than one copy of CED is running. The single port also makes it difficult for more than one program at a time to talk to CED.



```

CED View #3 ced.txt
resolution.
CED File: ced.txt
CEDPro review:
CED File: ced.txt
CEDPro review:

```

play was mangled in both file display and requesters. I hope a version compatible with 2.0 will be out soon.

The spiral-bound manual, by the way, is excellent. The well-organized, intelligible prose provides lots of useful, accurate advice. While head and shoulders above its peers, it could still stand a little improvement. For example, the tutorial asks the user to begin typing into the new view, which is impossible, because the default environment makes the file noneditable. This is hardly a reassuring start for a novice! I was glad to see an index, but it needs even more references (Autosave, for example, has only one page reference, and not to its major appearance).

In summary, CED is a solid program packed with useful features that will make any programmer's life easier. For my money, UEdit provides more features for the dollar than CED and a display that's more to my taste. I suspect, however, many loyalists and new converts will feel CED's Undo feature alone is cheap at twice the price.

925 Stewart St.
Madison, WI 53713
608/273-6585
\$99.95

Macro68

Algebra

Two new assemblers with new syntax, too.

By Jim Butterfield

able—some free, some supplied with compiler systems, and some sold commercially—a new assembler has to have excellent features to attract notice. Macro68 (The Puzzle Factory) is fast, full-featured, and covers a wide range of system and chip environments. As such, it's worthy of attention. The other newcomer, Algebra (Aelen Corp.), does not stand up as well to scrutiny.

GOOD NEWS FIRST: MACRO68

Macro68 is purely an assembler: It does not incorporate an editor or a full-scale linker. Commodore's MEmacs and The Software Distillery's BLink are supplied for doing those jobs. You might prefer another text editor, especially if you want to take advantage of the ARexx interface provided. No debug program is supplied, but the included profiler is useful in analyzing the efficiency of your code. For small or repetitive corrections, the supplied utility CHANGE90 allows "stream editing." If you wish to bypass the object/linking step and generate executable code directly, assembler directive EXEOBJ will let you.

The three-disk package is complemented by a 128-page manual and a large READ.ME file. Tutorial and reference materials are intermixed in both, which makes for heavy reading if you're looking for something specific, such as whether the program installs on a hard disk (it does). Regrettably, there is no index. You may need to read through the documentation more than once to get a feel for the scope of this package. Disk 3 contains numerous code examples.

Macro68 supports the Amiga's origi- ►

nal processor, the 68000, plus its big brothers: the 68010, 68020, 68030, and 68040. Good support exists for floating-point chips, the 68881 and 68882, and for the 68851 MMU; there's even provision to allow code to be written for the Amiga copper. The include files that come with the current version of Macro68 are 1.3 versions only. Version 2.0 include files will undoubtedly be made part of the package when Commodore releases and licenses them.

As the 68000 family has grown, Motorola has adopted a new system of syntax for opcodes and addressing modes. Prepare for a shock: Your existing program files and assembler will become partly outdated. Here are a couple of examples that show the kind of changes taking place. A favorite Amiga command, `MOVE.L 4,A6`, is now precisely coded as `MOVEA.L (4).w,A6`. In this case, the switch from `MOVE` to `MOVEA` and the addition of `.w` are not too significant—most assemblers would optimize these. The need for parentheses is a radical change, however, deemed necessary for uniformity with more complex addressing modes in the newer processors. Consider also the "short branch." Most programmers traditionally would write it similar to `BRA.S LOOP`. The `.S` flag for short is now reserved for the math coprocessor chips. In the new format, you code `BRA.B LOOP` using `.B` to signal a byte-length branch. Similarly, `BRA.L` becomes `BRA.W`, signaling that the branch is word-sized.

Macro68 does its best to adopt the new format without threatening the old. While assembling using the `NEWSYNTAX` option causes the assembler to insist on the new format, assembling with the `OLDSYNTAX` option (the default) instructs the assembler to recognize both the old and new formats. Alternatively, you could convert the old source to the new format via the `NEWSYNTAX` utility. A useful program, `NEWSYNTAX` runs quickly, does the job, and produces code with more detail. I have a tendency to write code such as `TST D0`; the conversion would change this to `TST.W D0`, reminding me that the instruction as written will test 16 bits, not the whole 32-bit `D0` register.

FAST AND FRIENDLY

Macro68 is as fast as any assembler I've

seen. A test file of 4000 statements, including 2000 symbols defined and used, assembled in less than seven seconds on my Amiga 2000. That's with modes `OLDSYNTAX` and `RELAX`, which are said to be the "slower" style of operation. The opposite modes, `NEWSYNTAX` and `STRICT`, however, produced the same results. Assembly time will, of course, be slower when you specify output files for listings or cross-references. As to listings, the format of these files is rather ragged and is not as detailed as I would like, but it's clear enough for most uses.

As Macro68's name suggests, it handles macros well. The existing standards are supported, and the manual outlines advanced uses in detail. Conditional statements and loops are easy to set up. You can even make preassembled macro libraries resident at assembly time, speeding up an already-fast assembly run.

Macro68 does little optimizing of code; only the most obvious things are done. Some assemblers, for example, change `MOVE.L #0,D0` to the appropriate `MOVEQ` instruction, or even discard `LEA 0(A4),A4` because it only wastes time. Not so with Macro68: What you write is mostly what you get. For example, backward branches are made short (byte-sized) if they reach, and `ADD` becomes `ADDA` or `ADDI` where appropriate. No subtle optimizations here.

You can customize Macro68 in many ways. While a configuration file lets you change defaults to your favorite settings, you can adjust more than assembler options. You can define symbols or even generate startup code if desired. A custom file, more correctly a customizing file, goes deeper, letting you change the workings of the assembler itself. For example, if you do not like or use the `ABCD` command, you can remove it completely from the assembler and save the lookup time, or you can move it down the priority list so that this code is only searched for after everything else has been checked. Suppose you use `MOVEQ` frequently—you can move this instruction to the top of the lookup table to gain extra assembly speed.

Macro68 is rich in directives. You can execute `CLI` commands during the assembly process or even send `ARexx` messages. You can quickly define structures by use of the `SO` (structure offset) directives, comparable to the `RS` directive of some other assemblers.

There's also a "negative `SO`" directive, `FO`, (frame offset) to allow easy stack frame definitions.

All these new features add up to make Macro68 a strong contender. Fast, powerful, and flexible, it's even up to date with the new Motorola syntax. Perhaps this assembler is not as user-friendly as some requester- and menu-driven ones, but it gets the job done quickly.

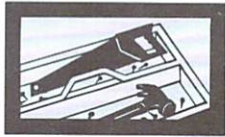
THE INDIVIDUALIST: ALGEBRA

By contrast, Algebra tries to be too friendly. An assembler-style program that produces code for an Amiga or a single-board computer, Algebra proposes a new format for writing 68000 code, quite different from the standard assembly language. For example, to add a value of 39 to register `D3`, you would use `D3+#39` as opposed to the conventional assembler coding `ADD.W #39,D3`. Similarly, if you want your program to branch if register `D5` contains a value of 10, you would use `cmp d5 #10 L` followed by `beq target` rather than the traditional `CMP.L #10,D5` and `BEQ TARGET`. Make your own decision as to which is easier. I'm used to the traditional forms, but Al DePaul, the originator of Algebra, obviously feels that the algebraic method is more natural.

The Algebra package consists of a single disk; the manual comes as a data file on the disk. No include files are provided. If you want to call Exec function `AllocMem`, for example, you must sort out for yourself that the offset is `-216` decimal. Algebra itself is actually two programs—`ALGEBRA`, the assembler, and `BLD`, the loader. Both were written in compiled BASIC and run reasonably quickly for small programs. They are not, however, in the same league as professional assemblers.

Algebra converts the original code into machine language. Not an object file, not an executable file, but pure machine language. As such, it won't load directly into the Amiga. You must use `BLD` for each load to make space for the program. `BLD` leaves a message in RAM stating what memory space it has reserved. Your "main" program then reads the message, reads the machine language into the reserved space, and calls it. That's uncomfortable, because the Amiga's built-in loader automatically finds space for a properly constructed loadable pro-

Continued on p. 40



TNT

Technical News and Tools from the Amiga community.

Compiled by Linda Barrett Laflamme

Boost Your Memory Power

If you can't memorize something, know where to look it up. To help you, Vidia (PO Box 1180, Manhattan Beach, CA 90266, 213/379-7139) has compiled the **Amiga Programmer's Quick Reference**. This 16-page, 8½ × 11-inch book provides a complete ASCII table (including binary and hex translations), ANSI console control and report codes, Rawkey codes, compiler and linker flags for Aztec and SAS (formerly Lattice) C compilers, library names and base addresses, Amiga con-

sole control sequences, the console device report stream, and guru meditation error definitions. Whether you need such general information as C or 68000 assembly language command explanations and guidelines or specific details, such as note frequencies or RGB color values, the Quick Reference is the place to look. Add to that the programming pointers in the margins and the \$7.95 price seems a bargain. Graphics and desktop publishing guides are also available.

A System Plus Source

Gear up for Unix on the Amiga with **MINIX 1.5** (\$169). Originally developed as a teaching tool, MINIX 1.5 is a Unix-like multitasking, multiuser operating system from Prentice Hall (College Technical and Reference Division, Simon & Schuster Higher Education Group, Prentice Hall Building, Englewood Cliffs, NJ 07632, 800/624-0023, 201/767-5969). Sys-

tem-call compatible with Unix version 7, the package includes a K&R-compatible C compiler, three editors based on ed, vi, and emacs; a shell identical to Unix's Bourne shell, more than 125 utilities (including cat, grep, li, make, nroff, sort, and spell), plus more than 255 library procedures (including atoi, fork, malloc, read, and stdio). MINIX 1.5 also comes with

the complete C source code for almost all utilities and the operating system. For those who prefer to read code in print instead of on the screen, the 688-page manual lists the entire operating system's source code. You will need at least one megabyte of RAM to run the nine-disk package, but a hard drive will do you no good. They are not supported.

The Rx for ARexx

ARexx let you combine your compatible programs to produce even more powerful applications. **Rx_Tools** (\$54.95) goes one step further, pairing with ARexx to let you access Intuition and add custom graphical user interfaces to your combined programs. The ARexx and Rx_Tools team gives you a complete development environment (with a built-in text editor) with which you can more easily add windows, requesters, menus, and gadgets to your programs. For more details contact TTR Development, 1120 Gammon Lane, Madison, WI 53719, 608/277-8071.

Any Way You Like It

TurboText (\$99) promises speed and flexibility at any level. You can simply record macros or use the interactive ARexx-based development tools to write, check syntax of, compile, and test code in any language. The basic functions list includes centering, justification, search and replace, upper/lower-case conversion, and clipboard-based cut and paste. For advanced features you can expect an integrated programmer's calculator, a hexadecimal editing window, outlining capability,

text template support, keyboard and menu remapping, plus the ability to redefine all text strings so you can customize the editor to your favorite language. Old editing habits can be hard to break, so TurboText emulates such Amiga, Unix, and PC editors as TxE+, CygnusEd, MicroEMACS, and QEdit. Need even more customization options? Take a look at the ARexx macro interface with more than 130 commands, auto-case-correction, which automatically switches letters to which- ▶

ever case you specify, and auto-word-correction, which lets you define a dictionary the program uses to automatically correct mis-

spellings. The company behind the product is Oxxi Inc.; contact them at 1339 E. 28th St., Long Beach, CA 90806, 213/427-1227.

Toaster Challenge

If you're one of the many who see the Video Toaster and say, "What great effects, but I would have programmed it differently," you're going to get a chance to prove your words. NewTek is releasing specifications on how to access the Toaster libraries, as well as a callable ARexx interface to the libraries. Stay tuned for more details or, for the latest information, contact NewTek directly at 215 E. 8th St., Topeka, KS 66603, 913/354-1146.

Eyes on the Summer

While still deep in development, Oxxi's **Modula II** compiler (\$199) is reportedly posting some interesting numbers. On a stock Amiga, this multipass compiler sped to a rating of 1500 Drystones, compared to SAS C's 1100-1200 Drystones for the same test. Promising to use the IEEE floating-point library and to handle blocks larger than 32K, Modula II is scheduled for release in June. Watch for more details.

On Tap

Whether you have a new product to proudly announce or some juicy news to whisper, we've got willing ears. Drop us a note at TNT, The Amiga-World Tech Journal, 80 Elm St., Peterborough, NH 03458, or call 603/924-0100, ext. 118.

PD Toolbox

Fancy packaging and splashy ads don't mean a product is superior, especially when you're comparing technical applications. Many plain-clothes PD programs deserve as much mention as their commercial cousins. Consider compilers, for example. For your first foray into a foreign language, experimenting with a PD compiler before investing in a commercial equivalent makes good sense. You may find you never need to switch.

On BIX alone, you'll find several choices for compilers, with C packages (such as **cc68K_c.arc** and the **sozobonc_1.zoo/sozobonc_2.zoo** combination) most common. Of particular interest are **DICE C** version 2.05.07 (**dice205c.lzh**) from Matt Dillion and **PDC** (**pdcbn.lzh**) from Lionel Hummel, Paul Petersen, and friends. **DICE** (Dillon's Integrated C Environment) is a complete C preprocessor, including a compiler, an assembler, a linker, and support libraries, plus a dme editor. For features you can expect ANSI compatibility, plenty of code optimizations, autoint routines, and more. You will, however, need **includes/amiga.lib** from Commodore. Housed in three files (**pdcbn.lzh**, **pdclsrc2.lzh**, and **pdclsrc.lzh**), the PDC package in-

cludes a compiler, an assembler, a linker, and a librarian, as well as several utilities, documentation files, libraries, and header files. PDC supports such features as all ANSI preprocessor directives, function prototyping, and structure passing and assignment, plus it also supports SAS C compatible libcall pragmas, precompiled header files, built-in functions, and stack checking codes. If you really want to dig into the system's guts, you can study all the source code, which is kindly provided.

Speak with a forked tongue? Check out **DevKit** (**devkit.zoo**) by Peter Cherna. A combination of C and ARexx language programs, DevKit lets you launch your compiler from within your editor, automatically position the cursor on errors, press one key to look up the autodoc page for any Amiga function, find a system structure or **#define** within the include files, or find any function in the your source code. You can also put the compiler options in the source code in a supplied utility for quick reference and modification.

More exotic languages are also available. Version 1.2 of the **DRACO** compiler (**draco_v12.zoo**) is a hybrid of Pascal and C

by Chris Gray designed to create fast and compact object code. It can access the majority of the operating system's features; see the file **DRACO_DOCS.ZOO** for more details. If you prefer a straight **Pascal** compiler, check out **pascal.lzh**.

Finally, the **Oberon** (**oberon.lzh**) is an object-oriented language developed by Prof. Dr. Niklaus Wirth of ETH Zurich in Switzerland as a successor to Modula-2. A single-pass compiler, this version creates standard Amiga object files, uses a multitude of optimizations, supports writing of re-entrant programs, and lets you call code from other languages. The package includes a compiler, an editor, a linker, a program to display compilation errors, as well as demos. You can thank author Fridtjof Siebert for this version.

The PD is no slouch on utilities, either. Four worth looking at are **KickDate**, **ParNet**, **RexxHostLib**, and **SetCPU**, found on Fred Fish disks 408, 400, 403, and 400, respectively. **KickDate** saves and retrieves the system's current date stamp to and from the first sector of the Kickstart disk. A1000 owners with autobooting hard drives will appreciate this program, as it saves the system time through reboots and power downs. All pro-

grammers will appreciate the source code that author Joe Porkka provides. With a special DB25 cable and ParNet (**parnet2.lzh** on BIX), you can connect two Amigas via their parallel ports, and one can mount the other as a device, reading and writing its files as if they were local. This version 2.4 is the Software Distillery's **NET:** file system that uses Matt Dillon's parallel port code. Written by Olaf Barthel, **RexxHostLib** version 36.14 (**rxholi.lzh** in CompuServe's AmigaTech Programmers Utilities library) is a shared library package that helps you create and manage ARexx hosts. So that you can control ARexx from programs in Amiga Basic and such, the package includes Rexx-message parsing. Yes, Olaf also provides source. An update of an old favorite, **SetCPU** version 1.6 (**setcpu.lzh** on BIX) lets you detect and modify parameters related to 32-bit CPUs with commands that enable and disable text/data caches, switch on and off the 68030 burst-cache line-fill request, use the MMU to run a ROM image from 32-bit memory, report parameters when called from a script, and more. Commodore engineer Dave Haynie even provides his source code.

The AmigaWorld Tech Journal Disk



This nonbootable disk is divided into two main directories, *Articles* and *Applications*. *Articles* is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas not 101MOBSIB. The remainder of the disk, *Applications*, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

All the supplied files are freely distributable—copy them, give them to friends, take them to the office, alter the source if it's provided. Do not, however, resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return it to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

Probing Your System's Current AmigaDOS Device List

Follow these pointers and structures to a system-wide list of active devices.

By Eugene Mortimore

ONE OF THE FEATURES that make the Amiga so powerful is its ability to maintain vital system-wide information on precisely-current linked lists. This theme runs throughout all Exec operating subsystems and is central to a complete understanding of this complex software system.

The Exec software system accomplishes its principal bookkeeping chores by continuously updating parameters in its ExecBase structure and that structure's List substructures. For instance, if a user launches a new program and that program then subsequently adds a new library to the system, the Exec system will recognize that event and automatically update the ExecBase structure library list placing the new library on the list.

At any time, from initial system creation forward, the Exec system software automatically assures that this summary data always represents a capsulized current-system-state snapshot. Then, by reading appropriate structure parameters, you can always probe this snapshot to determine which software resources are in the system at that time. Through this probing, each program can determine which Exec software resources are available to it at each point of execution and, as necessary, add any required software resources that are not already present. The hidden Exec internal routines allow you to ascertain crucial information whenever your program needs it.

The ExecBase structure and its List substructures allow programmers to probe into current lists for Exec libraries, Exec memory blocks, Exec software interrupts, and so on. The Exec system also maintains a list of Exec devices; typically a program would find the serial.device and the parallel.device and others on this list. These are device names in the Exec device name space.

The AmigaDOS system deals in a different device name space. Here, for example, SER: is the name of the serial device and PAR: is the name of the parallel device. Both of these AmigaDOS devices use the underlying Exec device; any reference to the SER: or PAR: device will always result in execution of the Exec serial or parallel device routine, respectively. Therefore, the system has a set of shorter-named AmigaDOS devices. It is then natural to ask the following questions:

- Does the AmigaDOS system have a device-related list-based software-resource bookkeeping mechanism similar to Exec's system?
- If so, how and when does the system update its AmigaDOS device list?
- How can a programmer work through the AmigaDOS device list to determine which AmigaDOS devices are currently in a specific system?

- Finally, how can a programmer use the AmigaDOS device list to advantage?

TYPES OF AMIGADOS SOFTWARE DEVICES

The AmigaDOS system deals in AmigaDOS software devices. By system convention, all AmigaDOS devices are always assigned to one of three device categories: *real* devices (related to actual physical devices), *logical assigned directory* devices and *disk volume* devices. The dos.h include file contains three C #define statements that name these categories as DLT_DEVICE, DLT_DIRECTORY, and DLT_VOLUME, respectively. (As an important aside, you should know the relationship—and distinction between—AmigaDOS devices and Amiga processes. Each AmigaDOS software device is always associated with *one-or-more* Amiga processes, each process being defined by its own AmigaDOS process structure. Thus while there may indeed be *more than one* active serial device process in the system at any one time, there is still only one AmigaDOS SER: software device. Its executable code is used by all of those serial device processes.)

Just as for Exec above, the AmigaDOS system maintains these AmigaDOS devices as separate device nodes on a system-wide list a program can examine at any time. By examining the current list, the program can determine in detail how a machine's hardware system is currently configured—which real physical devices are currently mounted, which logical directory devices are currently assigned, and which disk volumes are currently in the system's disk drives.

For example, if your program needs to determine if the hardware system has a third floppy drive associated with a mounted AmigaDOS DF2: software device—perhaps an MS-DOS 5 $\frac{1}{2}$ -inch drive—it can examine the current AmigaDOS device list for a DF2: device. Once DF2: is confirmed to be mounted, and therefore present on the AmigaDOS device list, additional program statements can then examine the device characteristics of DF2: to determine if it is a MS-DOS 5 $\frac{1}{2}$ -inch disk-drive device.

Similarly, the same list-searching procedure will allow your program to determine if the logical libs: and devs: (or other) directories have indeed been properly assigned and to determine if a specific disk volume is indeed currently present in a specific disk drive.

Disk volume verification is required because many complex multifloppy programs expect a specific disk volume (with a specific volume name) to be inserted in a drive before they can continue. Now, if that disk volume is indeed in the proper drive, the AmigaDOS system will have already automatically placed it on the current AmigaDOS device list. If

the disk is not in the proper drive, it will not be on the current AmigaDOS device list. The program can then alert the user to insert it in the proper drive.

REAL DEVICES

Table 1.0 summarizes some of the most common real AmigaDOS software devices. You can see that these devices are always related to physical hardware—hence the adjective real. Note that some of these (software) devices talk directly to the underlying Exec (software) device while some use an intermediate file call a *handler* file that is always in the system's L: directory; this handler file then talks directly to the underlying Exec (software) device. Some of these (software) devices are automatically mounted by the system

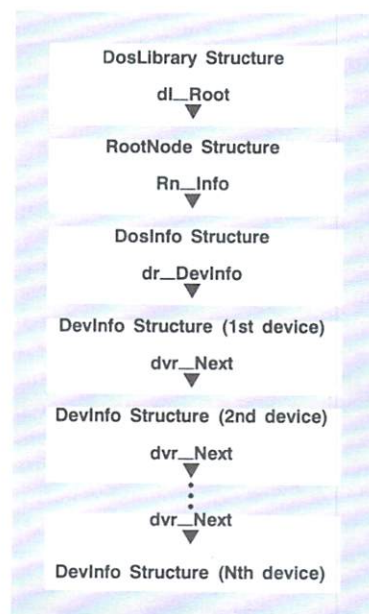


Figure 1. Relationship between bookkeeping structures in the AmigaDOS system software.

while others may require explicit devs: directory MountList file entries and related MOUNT commands.

In a minimally configured system you will typically find the following real AmigaDOS devices: DF0:, DF1:, SER:, PAR:, PRT:, RAW:, CON:, and RAM:. A more advanced hard-drive system perhaps configured for engineering purposes might have HD0: and HD1: devices and perhaps LPEN: (light pen) or JSTK: (joystick) devices. HD0:, HD1:, LPEN:, ▶

Table 1.0: A Partial List of Real (Physical) Devices In Many Amiga Systems

Device Name	Description	
Real Physical Devices that Use ROM-based or DEVS: Directory Exec Device Files		
DF0:	Disk in DF0: (Floppy Disk 0): Disk type "DOS", "KICK":	Uses Exec "trackdisk.device"
DF1:	Disk in DF1: (Floppy Disk 1): Disk type "DOS", "KICK":	Uses Exec "trackdisk.device"
DF2:	Disk in DF2: (Floppy Disk 2): Disk type "DOS", "KICK":	Uses Exec "trackdisk.device"
DF3:	Disk in DF3: (Floppy Disk 3): Disk type "DOS", "KICK":	Uses Exec "trackdisk.device"
FFS:	Fast File System: (Hard Disk): Disk type "DOS", "KICK":	Uses Exec "hddisk.device"
FAST:	Fast File System: (Hard Disk): Disk type "DOS", "KICK":	Uses Exec "hddisk.device"
DH0:	Disk Partition DH0: (Hard Disk 0): Disk type "DOS", "KICK":	Uses Exec "hddisk.device"
DH1:	Disk Partition DH1: (Hard Disk 1): Disk type "DOS", "KICK":	Uses Exec "hddisk.device"
RAD:	Recoverable RAM Disk RAM: Disk: Disk type "DOS", "KICK":	Uses "ramdrive.device"
Real Physical Devices that Use L: Directory Handler Files		
SER:	Buffered serial device:	Uses Port-Handler handler file in L directory
AUX:	Unbuffered serial device:	Uses Aux-Handler handler file in L directory
PAR:	Buffered parallel device:	Uses Port-Handler handler file in L directory
PRT:	Printer device:	Uses Port-Handler handler file in L directory
PIPE:	Input-output intraprogram communication:	Uses Pipe-Handler handler file in L directory
SPEAK:	Speech output:	Uses Speak-Handler handler file in L directory
CON:	Cooked keystroke console device:	Uses ConHandler handler file in L directory
RAW:	Raw keystroke console device:	Uses ConHandler handler file in L directory
NEWCON:	Enhanced replacement for CON:	Uses Newcon-Handler handler file in L directory
RAM:	Nonrecoverable RAM: disk: Type "DOS", "KICK":	Uses Ram-Handler handler file in L directory

and JSTK: entries would appear in that system's MountList file, and associated MOUNT commands would actually place these devices on the AmigaDOS system device list.

Going further, if you were so clever as to design the hardware for Carnegie Mellon's video harp, you would first create an Exec (software) videoharp.device to talk directly to your new video harp hardware and then create an l: directory-based VideoHarp-Handler (or whatever you wished to call it) file to talk to the underlying Exec videoharp.device. You would then place a VHRP: entry—with appropriate parameters describing the input-output characteristics of a video harp—in your devs: directory MountList file and issue a MOUNT VHRP: command. The MOUNT command would then place the VHRP: device on the AmigaDOS device list for any program to recognize and use as needed. That program would then be able to converse with the video harp.

The presence of any of these devices on the AmigaDOS device list implies the concurrent presence of hardware and controlling device software—the AmigaDOS device software—in the system. Generally speaking, then, real AmigaDOS devices represent software systems that allow process-information transfer back and forth between *actual hardware devices* and some other parts of the machine, most often, but not necessarily, RAM.

LOGICAL DIRECTORY DEVICES

The second type of AmigaDOS device present in all systems is the logical directory device. A typical system will include the following logical directory devices: SYS:, ENV:, DEVS:, LIBS:, FONTS:, S:, L:, and C:. Some of these devices are created by the system at bootup—for example, by hidden L:, C:, LIBS:, and DEVS: system assigns. Others are created when an ASSIGN command is directly executed.

For example, if your startup-sequence file contains many logical assigns, the system will automatically recognize these and create a separate AmigaDOS software device for each one you specify. Also, if a user explicitly enters an ASSIGN statement, the system will recognize that event and create a new logical directory device for it. If a program issues an EXECUTE("ASSIGN name1 name2") function call, the AmigaDOS system will place a logical directory device for name1 on the AmigaDOS device list.

DISK VOLUME DEVICES

The third type of AmigaDOS device is the disk volume device. In any system, each distinct physical disk will have a volume identifier assigned to that specific disk volume. This label is usually placed on the disk by the user with the RELABEL command. The AmigaDOS system will automatically recognize these volume names and create a software device for each

one, giving that device node the name of that volume.

BOOKKEEPING AMIGADOS STRUCTURES

Table 2.0 briefly describes the four structures—DosLibrary, RootNode, DosInfo (defined in the dosextens.h include file), and DevInfo (defined in the dos.h include file)—that the AmigaDOS software system uses to maintain its AmigaDOS device list bookkeeping. Figure 1.0 shows how specific instances of these four structures are linked together to allow your program to examine the current AmigaDOS device list. Let's take a look at the structure features that allow you to examine the most important AmigaDOS device names on the current AmigaDOS device software list.

The DosLibrary structure consists of an Exec Library (sub)structure with appended information specific to the AmigaDOS system:

```
struct DosLibrary {
    struct Library dllib;
    APTR dli_Root;
    APTR dli_GV;
    LONG dli_A2;
    LONG dli_A5;
    LONG dli_A6;
};
```

Your program can obtain a pointer to a DosLibrary structure instance as follows:

```
struct DosLibrary *dosLibrary=NULL;
dosLibrary=(struct DosLibrary *) OpenLibrary(DOSLIBRARYNAME, 0L);
```

Here dli_Root points to a specific RootNode structure instance.

The RootNode structure holds vital system-wide information about the DOS software system.

```
struct RootNode {
    BPTR rn_TaskArray;
    BPTR rn_ConsoleSegment;
    struct DateStamp rn_Time;
    LONG rn_RestartSeg;
    BPTR rn_Info;
    BPTR rn_FileHandlerSegment;
};
```

Your program can obtain a pointer to a RootNode structure instance as follows:

```
struct RootNode *rootNode=NULL;
rootNode=(struct RootNode *) dosLibrary->dli_Root;
```

The rn_TaskArray parameter maintains a current list of all active CLI processes, while the rn_ConsoleSegment param-

Table 2.0: A Summary of DOS Device Bookkeeping Structures: Description and Purpose

Structure Name	Description and Purpose of Structure
DosLibrary (1)	Exec library structure with appended DOS-specific data
RootNode	Defines system-wide information for all currently active CLI processes, disk-validator and file-handler processes
DosInfo	Defines Amiga computer network behavior and contains a pointer to a DevInfo structure
DevInfo	Defines a linked list for real, logical directory, and disk volume AmigaDOS devices

eter allows programs to probe into the details of all currently active CLI processes. Each CLI process will use the CON:, RAW:, or NEWCON: real device to define its executable code. The `rn_RestartSeg` and `rn_FileHandleSegment` parameters let you probe into the details of the generic (only one code copy used) disk validator process and the generic file handler process. Here `rn_Info` points to a specific system-initialized `DosInfo` structure instance.

The `DosInfo` structure helps define Amiga operations in a computer network. Its `di_DevInfo` parameter contains a pointer to the first `DevInfo` structure instance in the always current full system-wide device list.

```
struct DosInfo {
    BPTR di_McName;
    BPTR di_DevInfo;
    BPTR di_Devices;
    BPTR di_Handlers;
    BPTR di_NextEntry;
    LONG di_UseCount;
    BPTR di_SegPtr;
    BPTR di_SegName;
};
```

Your program can obtain a pointer to a `DosInfo` structure instance as follows:

```
struct DosInfo *dosInfo = NULL;
dosInfo = (struct DosInfo *) BADDR(rootnode->rn_Info);
```

Here `BADDR` is a `dos.h` include file macro that converts BCPL-language `BPTR` pointers to C-language pointers. The `di_DevInfo` pointer points to the first system-initialized `DevInfo` structure instance.

The `DevInfo` structure defines a linked list of C nodes for all three types of DOS devices. Each specific linked list structure instance represents one of these items in the current system. The system recognizes any device-creation events and continuously rearranges this list when such events occur. To show you how you can use this to advantage, the program below examines and prints key information about each device in your current system.

The definition of the `DevInfo` structure is:

```
struct DevInfo {
    BPTR dvi_Next;
    LONG dvi_Type;
    APTR dvi_Task;
    BPTR dvi_Lock;
    BSTR dvi_Handler;
    LONG dvi_StackSize;
    LONG dvi_Priority;
    LONG dvi_Startup;
    BPTR dvi_SegList;
    BPTR dvi_GlobVec;
    BSTR dvi_Name;
};
```

Your program can obtain a pointer to the first system-initialized `DevInfo` structure instance as follows:

```
struct DevInfo *devInfo = NULL;
devInfo = (struct DevInfo *) BADDR(dosInfo->di_DevInfo);
```

Your program can access deeper entries in the DOS device list using the system-maintained `dvi_Next` parameter as illustrated in the program below.

The `DevInfo` structure parameters pertinent to this discussion are:

dvi_Next a pointer to the next `DevInfo` structure instance in the list; NULL if this is the last device on the current DOS device list.

dvi_Type the specific type of device represented by this `DevInfo` structure instance: `dvi_Type = DLT_DEVICE (0)` for a real device; `dvi_Type = DLT_DEVICE (1)` for a logical directory device; `dvi_Type = DLT_DEVICE (2)` for a disk volume device.

dvi_Handler a pointer to the filename of a file that represents the handler for this AmigaDOS device. For example, for the real RAM device, this pointer will point to the name `RAM-Handler`, a file in the `L:` directory. (See Table 1.0.)

dvi_Name a pointer to the AmigaDOS device name of the device represented by this `DevInfo` structure instance. For example, the BCPL language `BSTR` representation of the string `DF0:`.

THEORY INTO PRACTICE

You can use the program in Listing.1 (in the accompanying disk's Mortimore directory) to generate a full current DOS system device list. You will notice that, apart from format, its output is very similar to the `ASSIGN` command's. You can therefore infer that the `ASSIGN` command examines the DOS device list in much the same way as Listing.1. To gain an understanding of the general type, context, and meaning of the program's output, execute the `ASSIGN` command in a CLI window and study its output before you run Listing.1 (which was compiled and linked with `Manx 3.6a`).

The program first opens the DOS (and ARP) libraries and then accesses all of the AmigaDOS device nodes in the current system list. It uses the ARP library `BtoCStr` function to convert `BSTR` strings to C strings. You must place the `arp.library` file in your `libs:` directory. You must also place the ARP library include files—`arplib.h` and `arplibfunc.h`—in your system's include directory. If the `arp.library` file is not present in the `libs:` directory, an Exec alert will appear on the screen. On the other hand, a failure to open the `dos.library` will cause the program to exit after first printing a message explaining the reason for failure. These two methods of reporting `OpenLibrary()` call errors illustrate two ways to alert the user to such problems.

The exact current contents of the DOS device list are defined by at least six different sources: DOS devices automatically mounted or assigned by the system at bootup; the current contents of your `devs:` directory `MountList` file and corresponding `MOUNT` commands to mount the devices in the `MountList` file; `ASSIGN` or `MOUNT` commands in your startup-sequence script file; explicit `ASSIGN` or `MOUNT` commands entered by a user in a CLI window; explicit `ASSIGN` or `MOUNT` commands executed by an `Execute` function call in a previously executed program in your system (since bootup); and disk volumes currently present in your system.

Typically, the system will always automatically create and mount `DF0:`, `DF1:`, `CON:`, `RAW:`, `SER:`, `PAR:`, and `PRT:` devices and perhaps several others at bootup. Also, the system will always automatically create several logical directories at bootup, including the `C:`, `L:`, `devs:`, and `libs:` directories. In addition, the startup-sequence file may, for instance, explicitly mount a

Continued on p. 46

Shared Libraries for the Lazy

*Want to build a library, but intimidated by
the bookkeeping? Try LibTool.*

By Jim Fiore

AN INTEGRAL PART of the operating system, shared libraries are collections of routines that all applications in the multitasking environment can use. Using existing libraries has always been easy, but building your own was a challenge. After many attempts, I have found a simple creation method. It does not require a lot of maintenance, and you can use it with a variety of languages, including assembly and C (both Manx and SAS, formerly Lattice). All you must create (and ever modify in the future) are the core functions and a function description file (sometimes referred to as an .fd file).

The example in the accompanying disk's Fiore directory turns a set of ordinary C functions into a shared library using the Manx C compiler, but you could use the SAS or assembly language instead.

A STROLL THROUGH THE STRUCTS

To better understand the program's workings, you should be familiar with a shared library's structure. Libraries are comprised of four main parts: a Library Node, a function jump table (often referred to as a vector table), the set of functions, and the global data for the library. In memory, a library looks similar to Figure 1. The definition of a Library structure is:

```
struct Library {
    struct Node lib__Node;
    UBYTE lib__Flags;
    UBYTE lib__pad;
    UWORD lib__NegSize;
    UWORD lib__PosSize;
    UWORD lib__Version;
    UWORD lib__Revision;
    APTR lib__IdString;
    ULONG lib__Sum;
    UWORD lib__OpenCnt;
};
```

The Exec keeps track of the library's status via the Flags field. The NegSize and PosSize fields hold the size (in bytes) of the library on either side of the library base. The Version and Revision fields indicate future changes and updates to the library. The IdString is a pointer to a null-terminated ASCII string that gives more information about the library. Exec uses Sum, the library checksum, to ensure library integrity. The OpenCnt field contains the number of tasks that have opened the library so far. Each time a task calls OpenLibrary() for this library, the library increments OpenCnt. Every time a program calls the complementary CloseLibrary() function, the field is decremented. If the OpenCnt

equals zero, Exec may remove the library to free more RAM.

A library's vector table is comprised of several six-byte entries, one for each function in the library. Each entry consists of a jump instruction (two bytes) followed by the absolute address of the function being called. Each function call then, is a multiple of six bytes behind the library base. For example, the seventh function in the table would be at location LibraryBase+42. These six-byte multiples are known as Library Vector Offsets or LVOs, for short.

In addition to the normal application functions, all shared libraries must have four mandatory functions—Open(), Close(), Expunge(), and Reserved(). Consequently, the first application function is always the fifth function in the list, at position 30 (referred to as the Bias, in a .fd file). The Open() and Close() functions are called for each OpenLibrary() and CloseLibrary() call. The Expunge() routine is used for final cleanup when OpenCnt equals zero and Exec has decided to remove the library. The Reserved() function is for future use. Presently, it should return a value of 0 only.

To properly load a library, Exec needs a Resident structure, commonly called a RomTag. In assembly, a RomTag looks something like this:

```
RomTag:
    dc.w$4AFC ;the RomTag identifier.
    dc.lRomTag
    dc.lendRom
    dc.bNO_AUTO_INIT ;Auto-initialize? In this case, no.
    dc.bVERSION ;Library version, as used in OpenLibrary( ).
    dc.bNT_LIBRARY ;Type. This is a Library.
    dc.bPRIORITY ;Not used.
    dc.lLib_Name ;Pointer to Name string.
    dc.lLib_Id ;Pointer to ID string.
    dc.lLib_Startup ;Pointer to initialization routine.
endRom
```

Normally, you set PRIORITY to 0. NT_LIBRARY is defined as 9, and the example program uses a nonauto-initialized library (NO_AUTO_INIT=0) to save space and reduce load time. The only hitch is that the RomTag must be in the first hunk of the file. If your development system will not allow this (for example, it requires code to be the first hunk and you cannot intermix code and data sections), you cannot make shared libraries. Versions 3.6 and earlier of the Manx C compiler suffered from this limitation, but version 5.0 does not.

When designing the library functions' code, you should keep a few rules in mind. First, the code should be re-entrant, meaning it does not use global variables. All variables should be held on the stack or allocated as needed, because several

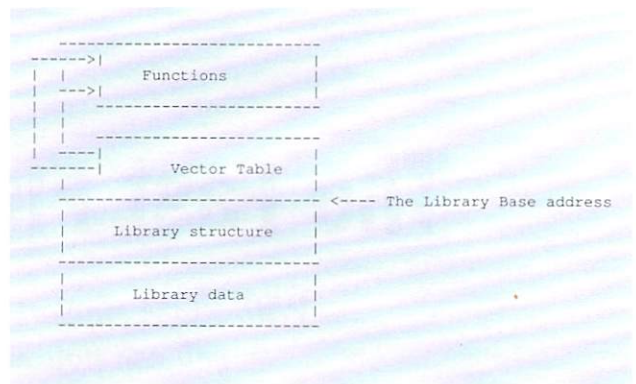


Figure 1. A typical library's structure in memory.

tasks can open and use the library simultaneously. If two tasks call the same function at about the same time, one task could alter a global variable before the other task is finished with it. The results would be chaotic at best. In contrast, you can use such global constants as fixed strings and look-up tables (for example, a sine wave table). Because constants are not altered by a function, tasks cannot step on each other. The second rule to remember is that if your library uses functions contained in other libraries, you must open and close these auxiliary libraries from within your library. For example, you could create a `DrawBox()` function based on the system functions `Move()` and `Draw()`. Because these two functions reside in the graphics library, you must open `graphics.library` when you initialize your library and close it as part of your library's `Expunge()` routine. Finally, do not use `printf()` style functions from within the library; it will not have access to `stdin`, `stdout`, or `stderr`. To get around this, the library functions should return error codes the calling program can interpret.

LET THE TOOL DO THE WORK

To build a library from scratch, you must create, compile, assemble, and link the `RomTag`, the Library structure, the function table, the initialization routine, the `Open()`, `Close()`, and `Expunge()` routines; and, of course, the core functions. While you can reuse some elements from library to library, you must still do a reasonable amount of setup work each time. To circumvent this, try the `LibTool` utility (in the accompanying disk's `Fiore` directory). Designed by Jeff Glatt of dissidents, `LibTool` creates all of the auxiliary items you need for a library (including pragmas and header files) from a single `.fd` file that you supply. Using `LibTool`, you need to make only two files to create a library: the functions of interest and the `.fd` file.

As an example, I am going to make a library that implements rectangular to polar and polar to rectangular conversions. The library will consist of four functions: `Mag()`, which accepts the real and imaginary rectangular components and returns the polar vector's magnitude; `Ang()`, which calculates the vector's angle in degrees from the rectangular components; `Real()`, which converts the polar magnitude and angle in degrees to the real rectangular component; and `Imaj()`, which transforms the same polar arguments into the imaginary rectangular component. (See `AWlib.c` in the `Fiore` directory.)

These functions require the use of floating-point math. For the sake of expediency, I am using Motorola fast floating point, so I must open `mathffp.library` and `mathtrans.library`

during initialization and close them as part of the clean up. In addition to the four conversion functions, `AWlib.c` contains `myInit()` and `myFree()`. The initialization routine calls `myInit()`, which opens the required math libraries and returns a `BOOL` value (`TRUE` or `FALSE`). If it cannot open the libraries, `myInit()` returns `FALSE`, which prevents my library from loading. Called by the `Expunge()` routine, `myFree()` closes the libraries that were initially opened. To use the `mathieee` libraries, my example library would have to open each `mathieee` library for each application that calls it. (Each task that uses `mathieee` libraries must open them itself thanks to the `mathieee` code that saves and restores the MPU context on task switches.)

Much of the drudgery is taken care of by `LibTool`, in conjunction with a specialized function description file. `LibTool` creates a library startup module containing the `RomTag`, the Library structure, the function table, the four mandatory functions, wedges into the various routines (the references to `myInit()` and `myFree()`), the proper version and revision numbers, strings, and so on. Also, this module will automatically open `Exec`, the `dos.library`, `intuition.library`, and `graphics.library`, because they are used so often (and are most likely already present in the system). If your library needs routines from these system libraries only, therefore, chances are that you will not need init and free routines. To aid in the construction of the applications that call your new library, `LibTool` also creates a header file, pragma statements, and C "glue" routines. The glue routines pull C arguments off the stack and place them in the proper registers for the library. `LibTool` can create libraries that expect arguments on the stack, as well. The resulting glue is smaller for C applications, but this does make it harder to access the library from other languages. Of course, you can bypass the whole issue of glue routines and use pragmas. By using pragmas, the C compiler moves the arguments into the registers and needs no glue.

The `.fd` file is an adaption of the Commodore-standard `.fd` files with which you are probably familiar. `LibTool` recognizes extra commands that allow it to create the complete library startup module. The example's `.fd` file follows:

```
##base AWBase *the name of our base
*used by C glue code and C PRAGMAS
##name AW *the name of our library (ie, aw.library)
##vers 1 *version #
##revs 0 *revision #
##init myInit *to be called once (upon loading the lib)
##expu myFree *to be called once (upon expunging the lib)
```

Continued on p. 47

The Fast Floppy System

*With ROM support for the FastFileSystem,
you can figure on faster floppies.*

By Betty Clay

IN THE BEGINNING, the Amiga had one filing system. The original AmigaDOS filing system is slow and cumbersome at times, but it has one great virtue—it allows you to recover almost anything lost on a disk. Every sector on the disks it formats contains a great deal of information to aid in file recovery.

As the file system matured, fewer disks failed and users clamored for faster drives. The result of Commodore's effort to meet this need is the FastFileSystem, or FFS. It was released two years ago, but was recommended for use with hard disks only—not floppies. The Amiga's ROM did not support it, and for that reason it was neither safer nor even much faster for use with floppies than was the old system.

Amiga OS 2.0 provides ROM support for both systems, so we can now safely use the FFS for floppies. To make wise decisions about using the new system, however, we need to study it. The differences, while not so great as to affect compatibility with disks we already have, are quite significant.

REGARDEZ LA DIFFÉRENCE!

To compare the two file systems fairly, I formatted one disk under each (using the 2.0 FORMAT command), and then copied the same information (Workbench 2.0 beta) to both disks. All the while, I observed their differences. I learned immediately that 2.0 can recognize a disk formatted under either system, while 1.3 is unable to validate FFS-formatted disks. For this reason, people who distribute disks widely will most likely continue to use the old file-system disks. AmigaDOS 2.0 defaults to the old file system, and you must add FFS to the startup-sequence's FORMAT command in order to format with the new system.

As a disk fills with data, the filing system creates several different kinds of blocks, or sectors. On current Amiga disks, each 512-byte block is divided into 128 longwords, called slots. There are various types of blocks (including boot blocks, directory blocks, file-header blocks, extension blocks, file-list blocks, and data blocks), some of which are not affected at all by the FastFileSystem. Let's take a look at those that are affected.

The first two blocks at the beginning of every hard or floppy disk are called the boot blocks, and the initial longword holds a number that indicates which filing system that disk uses. The old file system's number is 1146049280 in decimal form and 444F5300 in hexadecimal form. The ASCII translation of 444F5300 is: \$44=D; \$4F=O; \$53=S, and 00=0, for a result of DOS 0. Under the FastFileSystem, the numbers are 1146049281 decimal and 444F5301 hexadecimal, which translates to DOS 1.

The first thing the Amiga does is read the number in the first position on the disk. If the system does not recognize that number, the message "Not a DOS disk in dfn:" appears. If that number is corrupted on an otherwise valid Amiga disk, you may be able to rescue the disk by using a disk editor: Check the first longword, correct it if it is corrupt, and then correct the checksum before rewriting the sector to the disk.

DIRECTORY BLOCKS

After checking the boot block, the system moves to the root directory, which is the heart of the disk. The root directory is located in the center disk sector—sector 880 of a normal Amiga floppy disk. This sector begins with six slots of system information, followed by 72 slots of 32 bits each (occupying positions 6 through 77) that hold location data for user directories, icon files, file-header blocks, and so on. These 72 slots are called the keys, or pointers, to the files.

The restriction of 72 slots is not a system limitation; it is the current block size of 512 bytes that limits each directory block to 72 entries. The Amiga's software allows you to use blocks of other lengths, and through a system called hashing, you can enter more than 72 filenames in a directory. (Perhaps it is for this reason that the 72 slots are known as the hash table.)

Beginning with slot 78 there is a 32-bit word that holds the bitmap flags. This slot contains a 0 on an unvalidated disk and a -1 on a validated disk. If you insert an unvalidated disk (a disk that contains a 0 in this position) into your drive, the validator will automatically begin to read every byte on the disk and will rebuild your bitmap before placing a -1 in this slot. You cannot write to the disk (hard or floppy) until the validation process is complete—which can take quite a while, especially on a large hard disk filled with data.

Be sure to wait for a few seconds after the drive light goes out before removing a disk from the drive. Often, the light goes out initially because a file has finished writing, but then comes on again while the system updates the bitmap and sets the bitmap flag. Should you remove the disk before the bitmap flag is set, your disk will not be validated. The disk validator may be able to help you recover such a disk if you change the bitmap flag to 0 (unvalidated) using a disk editor.

The next 25 positions, numbers 79 through 103, are filled mostly with zeros on floppy disks, but it is here that a minor difference between the old and new file systems becomes apparent. The new system puts the first BitMapKey in position 103, the next in 102, and so on, and puts the BitMapExtension number (where applicable) in slot 79. The old system puts the BitMapExtension in position 103 and begins the BitMapKeys in position 102. On a floppy disk, there is no

need for more than one block of BitMapKeys, and there will never be a BitMapExtension number. Each position in the bitmap holds a 32-bit number, which signals whether 32 blocks are empty or filled, so a bitmap block can hold the data for 4064 sectors. Because an Amiga floppy has only 1760 sectors, one bitmap block is enough for any floppy—and even the new high-density disks do not have enough blocks to require more than one block for the bitmap. (This is not true, of course, for large hard disks.)

The bitmap block does not always remain in the same place on the disk. Its location is stored in position 79 of the root block, and on each of the disks I studied for this article, the bitmap was placed on sector 881—adjacent to the root directory itself. Each time you alter the disk, the new bitmap is written to a different place (the old one is not erased until the new one is safely stored). Its location data is then written into root-block slot 79. The remainder of the root block is unchanged between the two systems.

HEAD OF THE DISK

Although the FFS places icons nearer the center of the disk than the old system does, both systems read and use icons in exactly the same manner. The user directories and the file-header block (the first block of a file) are identical on both systems. The file-header block holds information that the system needs for file recovery, checksum, and so on, as well as the numbers of the sectors on which your file is stored. The first block your file uses is in position 77 of the file-header block, the second block is in position 76, and so on up to position 6—if the file is that long. Should it be longer, the file-header block will contain the number of an extension block that holds the numbers of the next group of sectors used by your file.

While the file-header blocks are identical in the two systems, the file-data blocks are very different. To demonstrate this difference, I have used a part of the DIR command from the C: directory. The commands are the same, copied from the same disk. Here are the beginnings of those blocks:

Old System	FFS	Explanation
0: 00000008		Kind of block (data)
1: 0000005C		Number of this sector (\$5C=93)
2: 00000001		How far into file? First block
3: 000001E8		Bytes of data per block (\$1E8=488)
4: 0000005E		Next block in this file (5E=95)
5: F4112B7B		Checksum
6: 000003F3	0: 000003F3	(hunk_header, which loads and

7: 00000000	1: 00000000	links the file into the system)
8: 00000001	2: 00000001	
9: 00000000	3: 00000000	
10: 00000000	4: 00000000	
11: 0000011A	5: 0000011A	
12: 000003E9	6: 000003E9	(hunk-code, the beginning of
13: 0000011A	7: 0000011A	the program)
14: 4E55FEA4	8: 4E55FEA4	
15: 48E73F32	9: 48E73F32	

As you can see from the hex dumps above, the files are identical except that the first six longwords have been removed in the FFS. Also, the FFS begins with the data in the first slot, while the old file system (OFS) begins in the sixth slot. This means that each data sector can hold an extra 24 bytes of information under the FFS. When I compared the storage on my two study disks, I found this:

DF2:	879K	1708	50	97%	0	Read/Write OFS
DF3:	879K	1648	110	93%	0	Read/Write FFS

Here you can see that the FFS saved me 4% of the floppy disk storage space and left an additional 60 blocks free (this is one advantage of using the FFS for floppies). The amount of extra storage space will vary. If there are many small files on the disk, the savings will be negligible, as each file has the same overhead. If there are a few large files, however, the extra storage space can be considerable.

Extra space is not the only advantage of the new filing system. When you load a file from the old file system, the system puts it into a disk buffer, strips off the 24 bytes of nondata, moves it into the user buffer, and then joins it to the remainder of the file. Under the FFS, this process is skipped altogether, and the data goes directly to the user buffer. Without that intermediate step, the computer can call for several contiguous blocks at a time, making an even larger savings in time.

There are some consequences, however, to using the Fast-File-System. These are mostly in the area of file recovery. Under the old system, the 24 bytes in the first six slots contained the position of this block in the file and the number of the block containing the next chunk of data. That information is helpful in rebuilding a ruined disk.

With the FFS, we are dependent upon the information contained elsewhere, although some of the information we need is still present. Each sector on the drive has a sector label that contains a sync mark to help the head position itself properly for reading. It also contains data on the drive's format type, the track and sector numbers, a checksum for the header and the file block, and 16 bytes that are reserved for future use.

Continued on p. 47

In Search of . . . The Perfect Joystick Routine

Looking for efficiency in all the "wrong" places.

By Rhett Anderson

ON THE AMIGA, there's usually a right way to program (the Intuition, multitasking way) and a wrong way. When programming to read a joystick, the right way consists of using the Gameport device. For assembly-language game programmers, however, the Gameport device is difficult to set up and read and, worse yet, it is much too slow.

Even if you're sworn to uphold the purity of the ROM Kernal Manual, it never hurts to know how the underlying hardware works. For that reason, I'm going to show you how to read a joystick the "wrong" way; that is, how to read the hardware directly. Although my examples are written in assembly language, the techniques I use may give you ideas for optimizing programs in your language of choice. As you will see by the final result, the wrong way of doing things can sometimes be right.

I'D RATHER SWITCH THAN SWITCH

Although the Amiga can use the proportional joysticks common on such machines as the Apple II, Tandy Color Computer, and PC compatibles, by far the most common kind of Amiga joystick is the type that originated on the Atari 2600 VCS game system and was adopted by Commodore. It offers four switches, each of which represents one of the cardinal directions: up, down, left, and right.

Older systems, including the C-64, can read the joystick simply by peeking bytes; each switch controls one of the bits. On the Amiga, however, the joystick is surprisingly difficult to decipher. The up and left switches control the two lowest bits in a byte, while the down and right switches occupy the two lowest bits in the neighboring byte. While you can read the left and right switches directly (a 1 means the joystick is pressed in that direction), reading the up and down positions is more complicated. You must use exclusive or (EOR) on the right and down bits to read the down direction, and on the left and up bits to read the up direction. (It never fails to amaze me what a hardware designer will do to save a few logic gates!)

When I first attempted to write a joystick-reading routine, I came up with Program 1 (see Listing.1 in the accompanying disk's Anderson directory). On entry, the routine assumes that the port number is in register D0. Most Amiga users keep a mouse plugged into port 0 and a joystick plugged into port 1 and will go to the trouble of plugging a joystick into port 0 only for a two-player game. For this reason, it is generally safe to assume that a one-player game will use port 1 for the joystick.

On exit, registers D0 and D1 each hold a value of -1, 0, or 1. D0 is used for horizontal values and D1 for vertical val-

ues. For instance, if the joystick is pressed down and to the left, D0 will be -1 and D1 will be 1.

CODE ON THE TABLE

If you value efficient, readable code, you will probably be as disappointed with Listing.1 as I was. The amount of branching involved is little short of absurd. There's plenty of room for improvement here (we could start by switching the MOVE.W immediate instructions to MOVEQ, or move quick, instructions), but a program like this needs far more than tweaking.

Assembly-language code such as Listing.1 begs for a lookup table. To build a lookup table for this project, I used the information received from the joystick as an index into a data table. My first problem was in finding a way to group the four important bits together. Here's what the data looked like when I read a word (16-bit) value from hardware location \$DFF00A (known as joy0port) or \$DFF00C (known as joy1port):

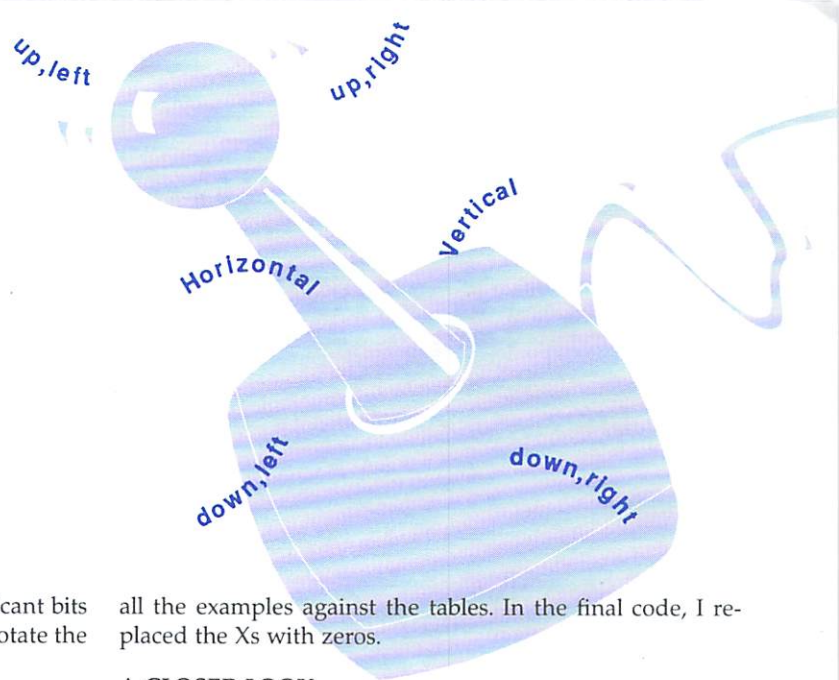
Value from \$DFF00A:
xxxxxxLUxxxxxxRD

(Note: Although the Amiga's hardware register base is \$DFF000, this value is not sacred. The proper way to get the hardware base is by using the symbol `_custom`, which is defined in the include file `hardware/custom.i`. As the include file says, do this to get the base of custom registers: `XREF _custom`. Directly accessing the hardware is not ideal, but most programmers agree that Commodore will not change the hardware base value—doing so would simply break too many programs. While I generally recommend playing it safe, for this example I'm throwing caution to the wind.)

Here, an x means that the bit has no relation to the joystick's position. Remember, up is really L EOR U, and down is really R EOR D. If you wanted to, you could turn D and U into the true down and left values with this section of code (from Listing.1):

```
move.w $dff00a,d0
move.w d0,d1
and.w #$202,d1
lsr.w #1,d1
eor.w d1,d0
and.w #$303,d0
```

In this example, I masked off the L and R bits, shifting them to the right, and used exclusive or to join them with the D and U bits. You do not really need to do this, however, as you can take the exclusive or patterns into account in the



lookup table. You *do* need to group the four significant bits together. After trying several options, I decided to rotate the lower byte two bits to the right, which gave me:

xxxxxxLURDxxxxxx

That's perfect, except that we will want to shift this value five places to the right so as to use it as an index into the tables. (Normally, you would shift it six places to the right, but here we need to look up the value of a table of words, not a table of bytes.)

With a debugger running, I held the joystick handle in all nine possible positions and watched the resulting values stream into the Amiga:

DIRECTION	LURD	DECIMAL
none	0000	0
down	0001	1
down,right	0010	2
right	0011	3
up	0100	4
up,right	0111	7
up,left	1000	8
left	1100	12
down,left	1101	13

This allowed me to build one table for horizontal positions and another for vertical positions. In the following tables, an X indicates a "don't care" condition; that is, a value that cannot be obtained with a standard joystick (for example, pressing the handle up, down, and right all at once).

Vertical dc.w 0, 1, 1, 0, -1, X, X, -1, -1, X, X, X, 0, 1, X, X
Horizontal dc.w 0, 0, 1, 1, 0, X, X, 1, -1, X, X, X, -1, -1, X, X

When using two lookup tables that include don't-care conditions, you can increase efficiency by embedding one table within the other or by overlapping the two. In this case, we can slide the horizontal table five words back into the vertical table to save ten bytes. Also, we can ignore the last two words of the horizontal table outright for a total savings of 14 bytes. Here's the result:

Vertical dc.w 0, 1, 1, 0, -1, X, X, -1, -1, X, X
Horizontal dc.w 0, 0, 1, 1, 0, 0, 0, 1, -1, X, X, X, -1, -1

Admittedly, this change took a bit of lateral thinking. If you are not convinced that this will work, just try it. Assume the joystick is pushed down and to the left. As is shown in the decimal-values table given us by the debugger, the index for this joystick position is 13 (13 words from vertical is 1; 13 words from horizontal is -1). You can verify this by checking

all the examples against the tables. In the final code, I replaced the Xs with zeros.

A CLOSER LOOK

Let's look at the final code piece by piece:

```
move.l #$dff00a,a0
add.w d0,d0
move.w (a0,d0.w),d0
```

This section of code moves the hardware value of the joystick into the D0 register. If D0 is 0, the value in \$DFF00A (port 0) is moved into D0. If D0 is 1, the value in \$DFF00C (port 1) is moved into D0.

Following the Amiga convention of treating registers D0, D1, A0, and A1 as scratch registers is a wise practice. If you use other registers in your routines, be sure to save them to the stack at the beginning of the routine and restore them at the end. Note that the last instruction above uses D0 as both an index and a destination. We will not need the port value after this instruction, and there's no sense disturbing other registers if you do not have to.

The following set of instructions merges the four directional bits together in the form 00000000000LURD0:

```
ror.b #2,d0
lsr.w #5,d0
and.w #30,d0
```

The ROR instruction groups the bits, while LSR moves them into reasonable numbers for an index, and the AND instruction masks out any unwanted bits (the value 30 is 11110 in binary). The possible results of this code are, in decimal: 0, 2, 4, 6, . . . 26, 28, 30. If you wanted to look up bytes instead of words, you would use ROR.B #2,d0; LSR.W #6,d0; AND.W #15,d0 instead. (In fact, you can cut the table size down by half by using bytes.) Then, before you return from the subroutine, you could use EXT.W D0 and EXT.W D1 to extend the byte values to words. This, however, would be a trade off of speed for size.

For the final section of the routine, I present two alternate sections of code:

```
lea Y,a0
move.w (a0,d0.w),d1
move.w 22(a0,d0.w),d0
rts
Y dc.w 0,1,1,0,-1,0,0,-1,-1,0,0
X dc.w 0,0,1,1,0,0,0,1,-1,0,0,0,-1,-1
```

Continued on p. 47

An Amiga Basic Graphical User Interface

*Bring your interface into the modern age with gadgets and requesters.
A few subprograms are all you need.*

By Bryan Catley

PRACTICALLY A "MUST have" for new programs, a Graphical User Interface (GUI) is merely a full-screen, point-and-click working environment. Text input, when necessary, is provided via specially designed requesters. Unfortunately, Amiga Basic provides no direct access to the operating system routines usually called to create a GUI.

Amiga Basic does directly support, however, many line drawing and graphics functions, the mouse, and a method of looking at each key press as it occurs. Why not use these readily available facilities to simulate the built-in routines that are so difficult to use from Amiga Basic? The basic idea would be to create a number of subprograms, each of which performs a predetermined task such as drawing a selection of gadgets, checking which gadget was selected by the user, accepting keyboard input from the user while providing full editing facilities, providing a text input requester, and so on. Yes, designing and setting up these subprograms requires some effort, but once they are available you can simply merge them into a program any time you need them, cutting subsequent development times considerably. (To save you even more time, I designed several for you already.)

To develop such subprograms, you need to store information about the (pseudo) gadgets, draw any specified range of gadgets into the current window, check which of a specified range of gadgets has been selected by the user and highlight that gadget, handle text input in a manner that gives the user all the necessary editing facilities, and implement a requester. You can find all of these capabilities via the readily available and standard Amiga Basic facilities.

PROCESSING GADGETS

Because the embossed gadgets are currently in vogue, we'll use this style in the example. The first thing we want to be able to do is store information such as the gadget's location, size, colors, and text. It will appear in the main program as DATA statements, and the first subprogram will read these and store the information in the appropriate of two arrays—one for numeric information and one for text. (Note that for the embossed style, you need three palettes, each containing a light, medium, and dark shade of the same basic color; this may mean you will need additional palettes which, in turn, may require the use of a custom screen.) This subprogram, called StoreGadgetInfo, will be used once during program initialization.

Next we want to be able to draw any specified range of these gadgets. This requires a subprogram that simply goes into the arrays and draws gadgets based on the given range and the contents of the arrays accessed. The main program

uses this subprogram (DisplayGadgets) each time a new window, which contains gadgets, is opened.

Finally, we need a subprogram that can check which of a given range of gadgets has been selected. The selected gadget should also be highlighted in some manner as long as the user keeps the mouse button pressed. In our example, we will change the embossed style to recessed for highlighting. This subprogram (CheckOnGadgets) will be used each time the program waits for the user to select a gadget.

Now, let's look at an example of each subprogram in depth. Check the Catley directory on the accompanying disk for a listing of each.

GADGETS

StoreGadgetInfo Of the three subprograms, this is by far the most straightforward. It simply reads the DATA statements defining the gadgets and stores the information in the appropriate arrays. The listing (GadgetSubs) shows the necessary format for the DATA statements. One point worthy of note is that the names of the gadget arrays are set up as shared variables. This means that once defined, the same names must be used at all times. (If you wish, you could use multiple arrays and pass the array names to each of the subprograms as additional parameters).

DisplayGadgets This subprogram simply draws all the gadgets within the range specified as input parameters. Note that DisplayGadgets uses the operating system routines Move& and Text& to position and display the text within the gadgets, letting you place the gadgets at any pixel location within the window. It also requires the use of graphics.library in the form of a graphics.bmap file in your libs: directory. If you are careful to always position the gadgets at multiples of eight, you may use the standard Amiga Basic LOCATE and PRINT commands, instead.

As coded, the gadgets should be no more than eight pixels high because of the method used to position text within the gadget. If you wish to use larger gadgets, you will need to modify the text positioning code.

The x,y + 6 in the positioning code sets the drawing point for the text string's baseline (assuming an eight-point font). If you use alternate fonts, you must modify the height of the gadgets and adjust the y + 6 for the new font's baseline. This may require additional parameters or shared variables.

CheckOnGadgets The most complicated of the three subprograms, CheckOnGadgets takes the mouse coordinates and compares them to the stored gadgets within the range specified. If found, the gadget is highlighted and the selected parameter is set to the relative number of the gadget

selected within the range specified. For example, if the range specified is 20 to 25 and the user chooses the 22nd gadget, the subprogram sets the selected parameter to 3. The highlighting remains until the user releases the mouse button. Note that there are three shared variables which play no direct role in the function of this subprogram. They are present for the potential use of the caller and other subprograms. For example, if a gadget is set up as a slider gadget (such as Preference's color gadgets), then not only do you need to know that the gadget was selected, but also where in the gadget the mouse pointer was clicked; `MouseX%` and `MouseY%` make this information available. The `MousePress` variable is used by the `UserInput` subprogram, which we'll discuss later.

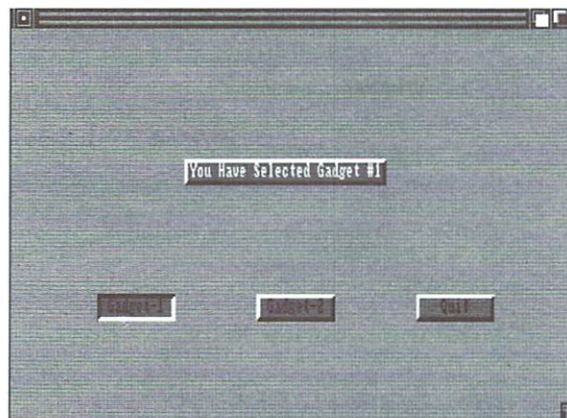
To see these subprograms in action, copy the `GadgetDemo1` program and `GadgetSubs` to your Amiga Basic disk, run `AmigaBasic`, load `GadgetDemo1`, and then merge `GadgetSubs`. Now, run the resultant program.

TEXT INPUT

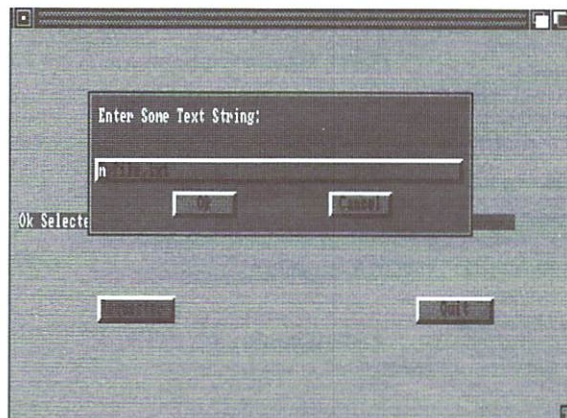
No matter how complete a GUI is, at some point the user will be required to enter some sort of specific textual information, such as a filename. When you look at the methods provided by Amiga Basic, however, you find not many are appropriate to a GUI. What we really want is a facility that provides, at a minimum, the capability to move a cursor back and forth without impacting the current text string, to backspace over characters (deleting them as it goes), to delete the character under the cursor, and to insert new characters at the current cursor position. Now, Amiga Basic does provide an `INKEY$` function that allows access to each key press. If we know where we are in the current window, we can combine `INKEY$` with a cursor that is nothing more than an 8×8 area which is placed in position, and subsequently erased (moved), with the `PUT (XOR option)` command.

Take a look at the subprogram named `UserInput` as an example. It also allows the user to erase the entire text string and start anew just by pressing the ESC key.

UserInput The necessary parameters are described in the listing. The major thing to note is that, like the gadget subprograms, `UserInput` uses the `Move&` and `Text&` operating system routines to position and display the text. Other than that, the subprogram simply waits for a key press (or an indication that a gadget has been selected), checks which key it is, and acts accordingly. If a gadget was selected, `UserInput` does nothing further and returns to the caller so the gadget selection may be processed. This is the purpose of the `MousePress` variable. For a demonstration of how this subprogram works, load `TextDemo1`, then merge `UserInput`



`GadgetDemo1` creates the gadgets and monitors mouse action.



`RequesterDemo1` combines button gadgets and an editable text gadget.

Sub, and run the resultant program.

PROGRAM REQUESTERS

Requesters are essentially a combination of prompts, (optional) text input, and gadgets all contained in a small window that opens when the user selects an option that warrants the use of a requester. Once again, the `Move&` and `Text&` routines display the prompts.

The major complication is the fact that the text input and the gadget checking routines must have some means of communication. For example, what if the user enters some input but clicks a gadget before pressing the return key; or what if the user presses the return key but then decides the string just entered requires modification? A requester must provide for all these eventualities. On the accompanying disk, you will find a set of two subprograms (`RequestData` and `YesNo`) stored collectively under the name `Requesters`.

RequestData asks the user for input and provides `Ok` and `Cancel` gadgets; the required input parameters are defined within the listing. The most important facts to note are that the colors used, the positioning of the window, and the maximum size of the text string to be entered or edited are somewhat arbitrary. Feel free to modify these values to suit your own preferences or those of your main program. Note that any modifications will probably require changes to the shared variables and possibly the `DATA` statement that defines the text input area. The positioning of the `Ok` and `Cancel` gadgets may also require adjustment.

YesNo This requester simply waits for the user to select a `Yes` or `No` gadget based on given prompts. Once again, the positioning and colors used are somewhat arbitrary and may

Continued on p. 39

The Amiga Zen Timer

*Speed up your assembly code with the help
of this software stopwatch.*

By Dan Babcock

WHEN WRITING TIGHT code on a tight schedule, you cannot afford to guess at optimizing it, merely juggling instructions based on intuition. Profilers are great for identifying the hot spots that consume the bulk of execution time, but they offer no help in fine-tuning these performance-critical chunks. You could consult the official instruction timings in Motorola's MC68000 manual, but the workings of a CPU are simply too complicated to be explained by a few rules or a table of numbers. True execution speed depends on the order and sequence of instructions, the state of the prefetch queue, DMA contention, and other factors. For the best results, you need to take real-world measurements.

VIDEO TIME

Programming one of the CIA timers is a solution, but an inaccurate one. These timers yield a resolution, roughly speaking, of ten MC68000 cycles. For superior results, use the Amiga's built-in video beam counter. This counter has a resolution of one color clock, which equals 279.3651148 nanoseconds, assuming your system has an NTSC-type oscillator (28.63636 MHz). Stated another way, it sounds even better: One color clock equals two MC68000 clocks. Because all MC68000 instructions take an even number of clock cycles to complete, the beam counter provides a perfect timer. This counter is very useful for measuring the speed of a blitter operation, as well.

Don't worry, the horizontal blanking period does not introduce a discontinuity in the beam counter. The beam counter increments in a uniform manner for an entire display frame. I have, however, observed quirky behavior at the end of a frame. To avoid this problem, start timing at the second scan line of a display frame.

Building a timer program around this counter is relatively simple. Find the x,y coordinates of the video beam by taking two samples of the beam counter, one just before the test code and one after. After the test, compute and display the elapsed time using the stored beam counter samples. This is easy to do, when you know that in NTSC mode one scan line equals 227.5 color clocks (in PAL, one scan line equals 227 color clocks). Multiply each y by 227.5 (227) then add the appropriate x to determine the number of color clocks. Finally, you subtract the overhead of the timer to yield the final result. (Take a look at zen.i in the Babcock directory on the accompanying disk for a complete listing.)

Using the timer to time itself seems paradoxical, yet that is exactly what you do. The idea is as follows: Start and stop the timer with no intervening code. If the code to begin and end a timing session introduced no overhead, the result

should be zero. It is not zero, of course, but is exactly equal to the overhead of the timer.

BEAT THE CLOCK

When using the Zen Timer you need to keep a few restrictions in mind. The macros ZTimerOn and ZTimerOff start and stop the timing, respectively. Using this technique alone, you cannot time a code sequence longer than about 20 milliseconds (the PAL display period). If the maximum time is exceeded, the program prints an error message. Secondly, because the ZTimerOn turns off all interrupts for accuracy, your code cannot depend on interrupts, and almost all operating system calls are off-limits. The whole purpose of the Zen Timer is to measure the performance of relatively short code sequences, so this should not present a problem.

To view the results, perform a JSR or BSR to ZTimerReport. ZTimerReport computes the time spent in the code sequence between ZTimerOn and ZTimerOff, subtracts the timer overhead, and prints the results, in color clocks, to the so-called "standard output," normally the CLI. You need not call ZTimerReport immediately after performing a ZTimerOff. You are encouraged to modify this routine to suit your needs and preferences.

Consider the example:

```
;An example use of the Zen Timer
;Assemble with Macro68
exeobj
objfile 'testzen'
include 'zen.i'
ZTimerOn
;Your code goes here.
nop
ZTimerOff
bsr ZTimerReport
moveq #0,d0
rts
end
```

When you assemble and run this example, it prints to the CLI:

2 color clocks.

This tells you the system took four MC68000 cycles to execute the NOP instruction, exactly as expected.

As I said before, the Zen Timer is capable of timing small code sequences, even only one instruction. There is one catch, however: The execution speed of the first instruction is dependent to a certain extent on the state of the prefetch

queue. As a result, some caution is required.

Now, try adding NOP instructions to the example code. Each NOP should add exactly two color clocks to the execution time, assuming that code is run in zero-wait-state fast memory. In chip memory, the timings will vary slightly with each execution. For accuracy, you should always run a timing test several times and then calculate the average execution time.

Let's turn to a more interesting example. Suppose your code accesses the CIAs (or another chip that is accessed in a similar way). We know that the CIAs are accessed on ten-cycle boundaries. How should you arrange the code to minimize the synchronization delays? Ask the Zen Timer. When you run the code below, you will discover that the system executes it in between 26 and 30 color clocks.

;CIA access example

```
exeobj
objfile 'testzen.cia'
include 'zen.i'
ZTimerOn
move.b ($bfe001),d0
move.b ($bfe001),d1
ZTimerOff
bsr ZTimerReport
moveq #0,d0
rts
end
```

Now try:

;CIA access example 2

```
exeobj
objfile 'testzen.cia2'
include 'zen.i'
ZTimerOn
move.b ($bfe001),d0
nop
nop
move.b ($bfe001),d1
ZTimerOff
bsr ZTimerReport
moveq #0,d0
rts
end
```

This routine adds two time-wasting NOP instructions. Surprisingly, the timing results are exactly the same as for the previous example! Add one more NOP instruction, however, and you add a full ten MC68000 cycles to the execution time. Upon discovering this in a real program, you should try to fill the eight cycle void with a useful instruction, such as an address calculation (adda.l d2,a1) that the routine may need later.

As you can see, the Zen Timer lets you stop guessing and start timing! Use it on your own "efficient" routines; you may be surprised by the results. ■

Dan Babcock is an electrical engineering major at Pennsylvania State University and an avid assembly programmer. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

Graphical User Interface

From p. 37

be modified. Any modifications may require changes similar to those described for RequestData above.

For a demonstration, load RequesterDemo1, then merge GadgetSubs, UserInputSub, and RequesterSubs, respectively, from the accompanying disk and run the resultant program. Note that a mouse interrupt routine is used in this demonstration, which is of particular importance should your program also include custom menus, (simply add a menu interrupt routine, along with the one for the mouse, to give you complete control over everything the user may desire). You may also want to add an additional shared variable to the UserInput subprogram to detect a menu selection when it is made. A better approach is to temporarily turn off the menu interrupt when a requester is invoked.

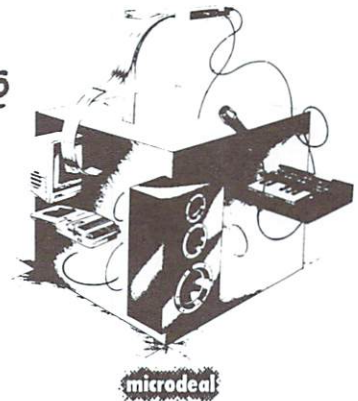
These six subprograms provide the basis for a simulated Graphical User Interface and may be used in virtually any Amiga Basic program, if you bear two things in mind. First, consider your own expansions to the capabilities described here. For example, modify UserInput to position the cursor directly over the character selected rather than always highlighting the first character. Second, if you use these subprograms with large programs that are not compiled, there will be noticeable delays the first time the various subprograms are used. Unfortunately, you cannot change this, it is just Amiga Basic rearranging memory for its own purposes. ■

Bryan Catley has been writing and programming for Amiga publications for many years. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.



...THE ADVANCED MIDI AMIGA SAMPLER...

\$199.⁹⁵



This stylish hardware unit provides your Amiga with full 8-bit stereo digitizing capabilities plus a full implementation midi interface - all in the same unit! The beautiful graphics/mouse software interface has dual real-time oscilloscopes and complete sound F/X manipulation tools including reverb and echo. The midi-support is provided in tandem with the digitizing software.

This is the software featured in the Paula Abdul video, "Cold-Hearted Snake." Check it out!



3201 Drummond Plaza
Newark, DE 19711
(302) 454-7946

From p. 22

gram. I was assured a supplementary Algebra package, not available at the time of this review, will construct such a loadable file.

REWRITTEN RULEBOOK

The documentation and examples provided on the Algebra disk break many of Commodore's style rules. Peeking and poking memory locations, other than address 4, is taboo by Commodore standards. Yet, Algebra happily suggests diving into hardware registers directly or passing values to and from subroutines by poking them into fixed locations. You *can* do all this, of course. It's your machine and you won't go up in flames the instant you break a rule. Normally, however, you should treat your multitasking Amiga with courtesy to avoid stepping on another task or building incompatibilities that will obstruct future system expansion. I do worry that beginners will start with these techniques before they understand the dangers. I wish the documentation and the examples were more conservative.

For me, Algebra is not a viable choice with so many first-class assemblers available. Its syntax is highly individual—too much so if you want to understand assembly language listings in books or magazines. If you can't grasp the standard method, however, Algebra will produce workable code and may be the starting point you need.

Macro68

The Puzzle Factory

PO Box 986

Veneta, OR 97487

503/935-3709

\$150

One megabyte required.

Algebra

Aelen Corp.

529 W 42th St.

New York, NY 10036

\$89.95

No special requirements.

The QPMA BASIC Programmer's Toolkit

In search of a better PSET.

By Robert D'Asto

SQUEEZING GREATER performance

from Amiga Basic graphics is a subject near to the hearts of many programmers. The language is easy and fun to use, but, unaided by lower level techniques or LIBRARY calls, it merely taps the throttle of the mighty graphics engine that lies below. With the Amiga's available palette of 4096 colors, BASIC provides direct support for no more than 32 at a time. This graphics performance gap creates a void for clever programmers to fill with tricks, tweaks, PEEKS, and POKES, as well as a variety of para-BASIC extenders, expanders, and supersets.

One recent solution is The QPMA BASIC Programmer's Toolkit from High Byte Software. Their claim, "allows Amiga Basic and AC/BASIC programmers to easily produce graphics in the new QPMA graphics mode, which features 320 × 200 resolution with 136 simultaneous colors from a total palette of 8.39 million colors [and] requires no special hardware," caught my immediate attention. Promising "color flexibility comparable to HAM Mode with none of HAM's problems" and "no complex screen slicing techniques," it fired my curiosity. All those features for only about \$30? This I had to see.

The QPMA package consists of a single, nonbootable disk without copy protection that contains the Toolkit, two Read.Me files of documentation, and several Amiga Basic and AC/BASIC examples of QPMA programming. The toolkit itself is an Amiga Basic source code "shell" into which you insert your own code. This shell holds the definitions of 17 Amiga Basic subprograms that provide the aforementioned graphics mode.

WHAT'S NEW?

The term QPMA, the documentation instructs, stands for "Quad-Pixel Matrix Array. . .and can be best described as a virtual graphics mode." The text goes on to explain how QPMA works. The author of this program discovered that placing two different-colored pixels next to each other on the screen produces the optical illusion that a third color, which is a combination of the two, is being displayed—sounds an awful lot like dithering to me. The documentation does later identify this technique as dithering and states, "Up until now. . .[dithering] has been hard

to program, and. . .there was no standard all-dithering graphics mode like QPMA. Dithering, for the most part, was practical only for digitizers. Until now." Really? I could have sworn I'd seen practical dithering in various graphics utilities from commercial paint programs to PD icon editors, and programming it in BASIC always seemed a snap via the PATERN and COLOR statements. But never mind that.

The text continues to explain that the QPMA system renders geometric shapes in four-pixel blocks on a hi-res interlaced screen. Each block contains two pixels of one color and two of another, in checkerboard fashion. This hi-res pixel block has an appearance similar to a single pixel on a lo-res screen, drawn in the color resulting from the combination of the two that make up the hi-res block. Because it uses a screen with a depth of four bit-planes, capable of producing 16 different colors, a total of 136 apparent colors are available when all possible pairs of the basic colors are dithered. In short, the QPMA system is dithering two colors at a time on a 640 × 400 screen in a manner that simulates the appearance of a 320 × 200 display, except that the programmer must now contend with the problems of flicker and the tiny text inherent in interlaced screens.

Where, then, is the promised new graphics mode, comparable to HAM, but with none of its problems? Reading through the documentation several times and studying the QPMA subprograms, which contain only standard Amiga Basic commands, I came to the conclusion that drawing graphics with QPMA is simply dithering on an interlaced screen and nothing more.

Dithering has been with us longer than the Amiga. Calling this a new graphics mode or even a virtual graphics mode and stating that it offers color flexibility comparable to HAM is simply misleading. I concede that dithering could be referred to as a graphics mode, but it is not new, and the simultaneous display of 136 apparent colors is in no way comparable to HAM's 4096 true colors. As for the total palette of 8.39 million colors, the docs explain that this is the number of possible two-color, dithered combinations of the Amiga's 4096 actual colors. Sorry, it doesn't work that way. A pal-

ette is the total number of *true* colors available, not *apparent* colors.

To write a QPMA program, you load the Toolkit into the Amiga Basic editor and type the source code into the space provided near the top of the file. You begin by setting up a 640 × 400 screen with the normal SCREEN statement and then drawing graphic objects by calling the various subprograms provided at the bottom of the file. The QPMA subprograms act as replacements for the Amiga Basic keywords MOUSE, PSET, POINT, LINE, CIRCLE, GET, and PUT. In addition, QPMASETBLOCK sets the desired dithering colors and the size of the basic "building block" of pixels used to render graphics. One remaining subprogram is used to determine the required size of a graphics array used for GET and PUT operations.

The routines do produce 136 apparent, dithered colors and are easy to use. Because no subroutines replace such keywords as PAINT and AREA, however, the graphics rendered with QPMA are limited to dots, lines, rectangles, boxes, and unfilled circles, as far as I can tell. You could use normal Amiga Basic keywords in addition to the QPMA subs, but, as the documentation states, their output looks odd when rendered next to the QPMA-dithered objects.

PUT ON A HAPPY FACE

I followed the instructions and wrote a program to render a single QPMA circle with a 50-pixel radius in the center of the screen. I first set the dithering colors with the QPMASETBLOCK routine and used the sub, QPMACIRCLE, to draw it. I then added code to calculate the elapsed time required to draw the circle. Finally, I signaled the interpreter to run my program, sat back, and watched the Quad-Pixel Matrix Arrays at work. After 44 seconds my circle was complete, smartly dithered in the two colors I had specified. I ran it again to be sure. Yes, it took 44 seconds to draw one, medium-sized, QPMA circle. Concerned with the rendering time required to draw a more complex design such as a happy face, I timed the rendering of a single horizontal line running the width of the screen: 15 seconds. At 44 seconds for a circle and 15 for a line, that's almost a minute without the eyes! I'm afraid I

never determined a final result for the happy-face benchmark, as I knew it would take longer than I was willing to wait.

The documentation file highly recommends compiling QPMA programs with Absoft's AC/BASIC compiler to speed up execution times. This method is about three times faster than the interpreter, but is still not fast enough to make real-time graphics rendering with QPMA practical. One of the examples provided on the QPMA disk is an AC/BASIC-compiled demo that displays a number of vertical bars that rapidly change color, showing off the 136 apparent hues available with dithering on a four-bit-plane screen. Drawing the bars takes a while, but, once rendered, they flash through their color cycles in admirable style. The trouble is the colors are changed via BASIC's PALETTE statement, not with any of the QPMA routines. The execution times for *drawing* graphic objects in all the examples were comparable to my own QPMA programming efforts.

QPMA is an interesting example of how simple, dithered graphics work. Other than that, I cannot think of a practical use for it. It may well have value to BASIC tinkers, but touting it as providing programmers with a new graphics mode having expanded color capability comparable to HAM is just not an accurate representation.

The QPMA BASIC
Programmer's Toolkit
High Byte Software
 91 Hillside St.
 Wilkes-Barre, PA 18702
 \$29.95 (plus \$2.00 S&H)
No special requirements.

FlexeLint 4.0

Sucks up errors like a vacuum.

By David T. McClellan

THE FIRST C compilers merely turned C into object code. They did not diagnose problems, which made finding code errors difficult. The situation worsened when C compilers began running on more than one machine and portability bugs cropped up faster than fungus in a refrigerator. So Kernighan and his team invented lint, a program that picks out problem bits of

fluff from a C program, such as function argument type conflicts, uninitialized variables, size conflicts in assigning pointers to ints, and using different data types for the same arguments to a function. Lint helped turn up many obscure bugs and also pre-tested code for portability.

FUZZ BUSTER

Gimpel Software's latest Amiga version of lint, FlexeLint 4.0, offers much more extensive C-fluff finding and highly configurable error-reporting. It runs fine on Lattice C and Manx Aztec C programs, and can find subtle bugs you introduced into code without noticing, especially the kind that don't show up under normal tests. As a cross-development plus, FlexeLint, using a large number of configurable code-checking options, can be made to examine C code as Microsoft C and PC or Unix lint does. If you carefully separate files with Amiga-specific parts from the portable parts, this option is a great help in catching porting problems early. Take a look at the following badly formed C program to get a feel for a few of the things FlexeLint can find.

```
#include <stdio.h>
#include <math.h>
#define XINC 1
#define DEBUGGING 1
#ifndef DEBUGGING
#define debmsg(x) printf("Debug msg: %s\n",x);
#endif DEBUGGING
main (argc, argv)
    int argc;
    char **argv;
{
    double dd, xx;
    int ii; FILE *out;
    if (! (out = fopen("outfile", "w")))
        exit(1);
    if (argc > 1)
        dd = 25.0;
    ii = sqrt(dd);
    if (ii != 5)
        debmsg("Bad sqrt!"); else
        fprintf(out, "SQRT of %d is %d\n", dd, ii);
    fclose(out);
}
```

Now, here is FlexeLint's report on the above mess:

```
#endif DEBUGGING

ex1.c 9 Warning 544: endif or else not followed by EOL
    if (! (out = fopen("outfile", "w")))  ►
```


ex1.c 19 Info 720: Boolean test of assignment

exit(1); ex1.c 20 Info 718: exit undeclared, assumed to return int

ex1.c 20 Info 746: call to exit not made in the presence of a prototype
 ll=sqrt(dd);

ex1.c 23 Warning 524: Loss of precision (assignment) (double to int)
 printf("SQRT of %d is %d\n",dd,ll);

ex1.c 27 Warning 559: Size of argument no. 2 inconsistent with format
 }

ex1.c 30 Warning 533: Return mode of main inconsistent with line 11

ex1.c 30 Warning 529: xx (line 15) not referenced

ex1.c 30 Info 715: argv (line 13) not referenced

— Wrap-up for Module: ex1.c
 Info 750: local macro XINC (line 4, file ex1.c) not referenced

— Global Wrap-up
 Warning 526: exit (line 20, file ex1.c) not defined

Warning 628: no argument information provided for function exit (line 20, file ex1.c)

As you can see, FlexeLint caught a number of subtle and not-so-subtle glitches. Some compilers don't allow anything after "#endif" (or #else for that matter). While the "...if (!(out = fopen(...)" is legal here, it could have been a mistyped version of "...if (!(out == fopen(...)", which would be a mistaken assignment. FlexeLint goes on to catch several more errors, including use of exit() without a prototype, a double (long real) being chopped down to an int (losing precision, possibly fatally), and a double being passed to a printf %d format. At the end it told me that I defined but never used "argv", "xx" and #define symbol "XINC"; "xx" and "XINC" are thus unnecessary (or placeholders for later). Most of these the Lattice and Manx compilers would not see, and would show up only when you tested the code.

"CORRECT" ERRORS

If you anticipate seeing certain errors often, because of your coding style or the fact that parts of your code are

young, you can direct FlexeLint to ignore them. For example, placing the command —e529 on the lint command line tells FlexeLint not to report error 529 (unused variable) for any file it processes. Alternatively, you could tell it to ignore that error (or any other) for a specific file or symbol, as in:

—efile(529,fred.c) (don't report it for file fred.c)
 —esym(529,xx,yy,zzzz) (ignore for variables xx, yy, and zzzz)

Because FlexeLint can process many files, in sequence, on its command line, you could also turn off an error before one set of files, then turn it back on for another, as in:

lint —e529 fred.c barney.c wilma.c +e529 betty.c bambam.c

FlexeLint will ignore classes of messages as well as individual ones. For example, —eau tells FlexeLint to ignore cases in which a function expects an unsigned number and you pass it a signed one.

When you wish to ignore certain individual errors or classes of reports every time you run FlexeLint, you can create a directive file with a text editor. Directive files contain —e<number> and other commands on separate lines and have names ending with .lnt. To use such a file, type its name on the FlexeLint command line before the C files it affects, as in:

lint MyIgnores fred barney wilma

To save you even more typing, FlexeLint tries appending first .lnt and then .c to filenames that don't have extensions. The program would read the names in the above command line, therefore, as MyIgnores.lnt, fred.c, barney.c, and wilma.c, respectively.

Directive files are also useful for explaining compiler eccentricities to FlexeLint. To begin a port to Microsoft MS-DOS C, you would create a micro.lnt file describing handling of near and far pointers and so on, and then use FlexeLint to check your code.

EXCLUDING INCLUDES

Unused defines and variables appear all the time in include files, because programs typically use only a subset of an include's contents. This would cause a lot of 529 and 750 errors, among others, if FlexeLint did not treat include files specially. When you enclose a filename in angle brackets

(<>), FlexeLint considers the file to be a "library" file and, unless told otherwise, ignores symbols that are defined but not used. FlexeLint provides directives for you to specify in great detail which files or directories of files should be treated this way, but the default case works fairly well. It searches for include files in the directory specified by the —i directive just as Lattice and Manx do. It will properly process not only <stdio.h> and similar "standard" include files, but also such Amiga-specific ones as <intuition/intuition.h>.

This brings me to FlexeLint's minor problem. Lattice users, such as myself, keep the include files in compressed form, to speed their processing by the compiler. FlexeLint cannot yet handle these compressed files (Gimpel's phrasing, so maybe in the future it will). So you must keep your Compiler-Headers floppy handy and pop it in when you use FlexeLint. Also, FlexeLint will use the Manx-style environment variable INCLUDE (set with Manx's set command) to find include files, but not the INCLUDE: device set by the ASSIGN command for Lattice to use. You can work around this, however, with a sequence such as:

ASSIGN IN: Lattice:Compiler-Headers
 lint —IIN: ...

or by putting the —i directive in a .lnt file. The string following —i, by the way, is concatenated directly to include file names, to avoid operating system-specific filename constraints. Put slashes at the end of your include directory names if you specify them on the command line.

CORRECTING THE TEACHER

I ran FlexeLint over a few of the programs I wrote for tutorial articles. I wish I'd had FlexeLint to use earlier; it turned up a few things in those programs that should not have seen print. If you work on large amounts of C code, particularly for projects you hope to port to other platforms, FlexeLint 4.0 is a must. ■

FlexeLint 4.0

Gimpel Software

3207 Hogarth Lane

Collegeville, PA 19426

215/584-4261

\$98

No special requirements.

Do-It-Yourself Software Marketing

*Staking your claim as a software entrepreneur requires more than
a good concept and a fast-running program.*

By John Foust, with Harriet Maybeck Tolly

WHETHER YOUR DREAM is to found the next Microsoft or just to turn a tidy profit on your after-hours programming projects, you face the same obstacles. In retrospect coding your application or game may seem easy compared to handling the marketing, packaging, advertising, and distribution. To map out the real costs and concerns of the software business, let's start a one-product company called Little Garage Software. To make it easy, we'll assume that you've already written and thoroughly beta-tested the program.

The first thing you have to decide is how many copies you think you can sell, so you know how many to make. Consider the limiting factors: If your product does not compare favorably with similar products in price and performance, sales will suffer. If it's a programmer's tool, or appeals only to CLI users, then the market is much smaller, but the program may sell at a steady rate for a long time. If it's a game, sales will be brisk for a few weeks, and then drop off to nothing.

Brashly assuming your product is positioned correctly and fills a market need, let's look at typical sales figures for Amiga products. The total number of Amigas sold puts a cap on our estimate. While Commodore proudly states that a million Amigas have been sold worldwide, that is over a five-year period and split among the different models. Commodore made approximately 150,000 Amiga 1000s. How many of the million are A500s, and how many are A2000s?

Estimating that only about 30 percent of the Amigas were sold in the United States. The best you can hope for is 300,000 sales, unless you also market your product in Europe and Canada. If the product needs a lot of memory, a hard disk, or a specific version of the operating system, the market shrinks again. As a comparison, early sales penetration of Electronic Arts' wildly successful DeluxePaint was once estimated at 30 percent of all Amiga owners, but a very popular Amiga product usually sells only about 10,000 copies in a year. A moderately popular product sells 3000 to 7000 copies. For a conservative estimate, plan to move 2000 copies.

GOING IN THE HOLE: PRODUCTION COSTS

The second most important part of the program package is the manual. Hiring a professional writer to create a small, 30-page manual costs around \$1000, so save money and write it yourself. You can design and print a bare-bones manual on a letter-quality printer with a simple word processor, but desktop-publishing software and a laser printer bring better results—at a greater cost, of course. Let's assume you're lucky enough to have access to the best printer and software.

Your printer is fine for one copy, but what about the other 1999? Offset printing runs at about three cents a page in vol-

ume. Let's say the manual is 30 pages long, and each page is half the size of a regular sheet of paper. Printing a half page on both sides costs the same as printing a full page on one side, so duplication for the manual comes to about one dollar apiece, plus 15 cents for a printed cardboard cover and two staples in the binding.

For packaging, simple one-color cardboard sleeves cost about 30 cents each. As long as you have the desktop-publishing software, you can design and print these yourself, as well. If you have your heart set on 4-color boxes, be prepared to shell out \$1.75 apiece for preparation and printing.

With two drives and a copy program, 2000 disks will take quite a while to duplicate, and then each one must be labeled. Custom-printed labels cost about 15 cents each. Disk-duplication services charge about 25 to 50 cents a disk for copying and labeling. Hire the service.

To shrink-wrap the packages with the manuals and disks inside, an outside company charges close to 25 cents a package (\$500 for 2000). If you do it yourself, shrink-wrap machines cost \$300 or so, plus plastic. At a rate of one minute per package, you will finish in a minimum of 33 hours.

Adding 50 cents for such incidentals as printing a warranty card and driving around town on errands, the cost for parts comes to \$3.05 per package. For 2000 copies, then, you need at least \$6100 to start Little Garage Software, and you will not recoup that for at least several months.

A good rule of thumb is that the retail price should be four to five times the production costs. With a raw-parts cost of \$3.05, the price would then be \$12.20 to \$15.25. You haven't paid yourself yet, however, or amortized fixed costs over the product's lifetime. For the sake of argument, let's set the program's retail price at \$49.95. Now you need to find people who will pay that.

THE BIG SPLASH: MARKETING

Advertising is a must, but it's not cheap. The most inexpensive national Amiga magazines charge approximately \$1500 for a full-page black-and-white ad and \$600 for a quarter-page ad. The circulation leader charges about \$1500 for a quarter-page ad, a higher rate because it reaches more people. Color ads are roughly \$6500 a pop, plus another \$500 for color separations. Before publication, someone must write the ad, compose it, and supply camera-ready art to the magazine. For color, hire a professional. You could make a small, black-and-white ad yourself, but small ads cannot carry much information and do not have the eye-catching appeal of full-page color.

Afraid you can't afford to advertise? Editorial coverage in ▶

magazines puts your product's name before the public, too. To get the program in front of the editors, you must write and print a press release to announce the product, and then mail at least 100 copies. You should also send out a dozen or so review copies, but this gambit can backfire. A good review will enhance sales, but a bad review could permanently damage them. In either case, you won't see the review until two to six months after the magazine receives the product.

Trade shows are a more immediate—but expensive—way to alert the public. If you have a superior product, you might get a small space in Commodore's booth at a show such as COMDEX. The chances are, however, that space is limited and your product isn't flashy enough. A booth at a recent AmiEXPO (now called AmigaWorld Expo) cost \$1600, plus \$300 for renting furniture, carpeting, and electricity. You'd need to compose and print flyers and press releases, box up Amigas and monitors, make decorations and signs for the booth, and ship it all to the show. Once the equipment is there, the labor union charges to move it to your booth. Now add the cost of airline tickets, downtown hotel bills and taxi service, and you could easily sink \$4000 into the event. If you're lucky, sales at the show will recoup these costs.

Imagine all the other costs of doing business: a separate phone line for arranging production, distribution, and press coverage (not to mention technical support), a database for invoicing distributors, dealers, and direct-mail sales; an answering machine, office supplies, postal scales, postage meters, letterhead, business cards, Federal Express and UPS accounts. Where will you put all this stuff? Business supply stores sell mailing envelopes and such in lots of 1000, not 50; you'll need a chunk of cash up front and lots of storage space.

Don't forget government paperwork, either. You need a local business permit, a state sales tax number, a federal tax ID number, and local sales tax permits for every trade show where you sell the product. The frequency of filing sales-tax returns depends upon the volume of sales in a particular state. With all this money and paper flowing around, you should hire an accountant and lawyer. Open a separate checking account, as well, to make sorting out the business taxes easier.

To pay yourself, you must file quarterly Social-Security and income-tax reports. To minimize liability, it would be better to opt for incorporation, as opposed to sole proprietorship. Incorporating, however, increases the cost of filing taxes, and the cheapest possible incorporation still runs to several hundred dollars. For further protection, you should trademark your program's name and search for conflicts with other product names, at a tune of several hundred dollars per name.

RING IT UP: SALES

You will take in the most money (almost \$40) on a direct sale to a customer, but few people buy straight from the manufacturer. That \$40 is not pure profit, either. For each direct sale, you need to make invoices and keep records, plus pack and ship it. This time and effort mounts up quickly, especially when you add the direct-sales payment hassles. What if a check bounces? How do you collect it? Do you delay shipment until the check clears? Should you mail C.O.D.? Getting credit card privileges from a bank is very difficult. In many states, you cannot legally accept credit card orders unless you are a corporation with a customer store front.

Selling through a distributor is more common and advan-

tageous. One shipment to a distributor replaces dozens of small shipments to dealers and users, getting the production run out of the garage faster. Similarly, the distributor collects money from dealers and pays you with a single check. Don't get too excited; you won't ship all 2000 copies at once. Distributors might order 50 or 100 copies at a time to fill their requests from local dealers.

You may also have to wait for your money. Distributors often demand credit terms from you of 30 to 60 days, meaning your check arrives five to nine weeks after you ship the product. Be warned: Developers love to swap horror stories about distributors who do not pay on time. Insist on C.O.D. payment instead. Of course, distributors balk at C.O.D. because it affects their cash flow.

While the distribution method reduces your money hassles, it also reduces your profit. Software passes through many hands before it reaches the dealer, and everyone along the way makes money. You, the developer, sell to distributors, who sell to dealers and mail-order houses, who in turn sell to users. A distributor typically buys software at 60 percent off the list price, so they pay \$19.98 for your \$49.95 product. Some distributors sell to other distributors for a small percentage over their price. A US distributor might sell it to a German distributor for about \$25. Some mail order firms buy direct from developers, which explains their low prices.

A local Amiga dealer pays about 50 or 60 percent of the retail price, or about \$30. Most dealers don't charge full retail prices, so their profit is \$13 to \$20. From this raw income, they must subtract their operating overhead.

For each copy sold, you take in about \$15, while everyone else gets about \$15, too. Over the course of the lifetime of this product, you'll get about \$30,000 if you sell all 2000 copies. You can use this to repay the initial investment and for your first paycheck. Meanwhile, the bills for advertising, telephone, professional services, office supplies, and printing arrive in a steady stream. Because the size of the Amiga market and the appeal of the product have placed a cap on our potential sales, Little Garage Software can't make more money in volume without raising the product price, which could in turn reduce sales.

BUSTLING OR BUSTED?

While this story had plenty of realistic detail, we also waved our hands and minimized costs. Little Garage Software wasn't a slick operation. It had no full-color ads in magazines, no fancy colorful packaging, no plastic boxes for the software. Frankly, it was a shoestring operation, and the software packaging looked cheesy. You can understand why many small software companies are short-lived. High start-up and fixed costs can eat away quickly at a small investment and force you back to a regular job.

The story of Little Garage Software shows why third-party developers are the lifeblood of any computer company. The entire chain of developers, distributors, and dealers collapses if companies aren't profitable. If the Amiga market is to have a professional image and professional software, customers must be willing to pay this price. ■

John Foust and Harriet Maybeck Tolly are president and vice-president, respectively, of Syndesis Corporation, which bore a suspicious resemblance to Little Garage Software at its inception. Contact them c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

Step Up To The Podium!

Admit it. You're an expert. You know how **it** works better than (almost) anyone. When you write code, you play by all the rules, yet your programs consistently out perform even those of the most wild-eyed ROM-jumping hacker. It's been obvious to you for some time that you should sit down at the keyboard, fire up your trusty text editor, and write an article explaining exactly how and why **it** should be done **your** way.

If the above description seems to fit you to a T, perhaps we should be talking. *The AmigaWorld Tech Journal* is looking for technical writers who have expertise in one or more areas and have a burning desire to share that information with the Amiga technical community. We need experts in all aspects of programming the Amiga, from operating systems to graphics to the Exec. You can write in any language you like—C, Assembly, Modula II, or BASIC. Best of all, you can include as much source code as you need, because all source and executable is supplied to the reader on disk. We also need hardware maestro's who can explain—in thorough detail—the inner workings of such complex components as the Amiga's chip set, expansion bus, and video slot. Don't forget algorithms either, we'll help you pass on your theories and discoveries.

The AmigaWorld Tech Journal offers you an unparalleled opportunity to reach the Amiga's technical community with your ideas and code and to be *paid* for your efforts as well. So, whatever your "**it**" is that you want to write about, *The Tech Journal* is the place to publish it.

We encourage the curious to write for a complete set of authors guidelines and welcome the eager to send hardcopies of their completed articles or one-page proposals outlining the article and any accompanying code. Contact us at:

The AmigaWorld Tech Journal
80 Elm St.
Peterborough, NH 03458
603/924-0100, ext. 118

From p. 12

handler (not a server), register a0 contains the base of the custom chips, a1 is the IS_DATA (which points to the appropriate channel's LoopStruct's loopLength), and SysBase is in a6. First, audioInt() checks if bit 31 of loopLength is set. Because this is always clear upon the first block-done interrupt, audioInt() does not turn the channel off—a good thing because the oneShot portion just started playing.

If the wave has a looping portion, program control falls through to I2, where the loop portion's location and length are written. Note that I am doing this even as the oneShot is starting to play, because the DMA has already copied the oneShot location and length to the backup registers. I do not, however, change the volume or period because the new values would take effect upon the next data fetch. Note that not only do I clear the interrupt (via intreq), which you should always do, but I also disable the block-done interrupt (via intena). This means that audioInt will not be called again when the DMA starts playing the looping portion. Instead, the loop will repeat until I turn the channel off in main.

If the wave has no looping portion, the program branches to I3 where it sets bit 31 of loopLength and does *not* disable the interrupt. This means that audioInt will be called again when the oneShot portion finishes, at which time bit 31 of loopLength will be set. The program then branches to I4 where it stops the channel. The net result is that it plays the oneShot portion once, and then stops the channel. Otherwise, the DMA would continue looping the oneShot!

Note that in _main, I make two calls to PlayNote(), once to play a wave with oneShot and looping portions and once to play a oneShot only. After I play the looped wave, I must stop the wave's playback, either by playing another note on this channel or calling StopChan(). I do the latter at label M7 in _main. Of course, you can play up to four waves simultaneously by calling PlayNote() four times with different channel numbers. Beyond this, PlayNote() will "steal" the channels from previously playing waves. At label M10, I make three calls to PlayNote() (each on a different channel) so that I sound a major triad. Note how I use TransposePeriod() to get the pitches of the chord.

FURTHER IMPROVEMENTS

To increase efficiency, instead of calling TransposePeriod() before each call to PlayNote(), you could make your own lookup table for each wave. Simply call TransposePeriod() for each step of the scale, stuffing the returns into an array of shorts, then use your step value to look up the appropriate period. This eliminates the lengthy division in TransposePeriod(), either method is more efficient than the audio device.

If you need to play sound effects for your application (typically oneShots), then you can use PlayNote() as is, and it will be much more efficient than using the audio device. If you want to play musical passages, you should make a higher level function that has a clock that tells you when you should play and release each note at some tempo (when to call PlayNote() and then StopChan()). No matter which implementation you choose, it will be more efficient than using the audio device. ■

Jeff Glatt is co-founder of and a principle programmer at dissidents, which specializes in music software. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

From p. 29

DF2: device—a 5 $\frac{1}{2}$ -inch MS-DOS drive—and a RAD: (recoverable RAM disk). Logical assigns to rename DF0:, DF1:, and RAM: also typically occur in a startup-sequence file; A:, B:, and R: are often common choices for these.

As we have seen above, the DosLibrary, RootNode, DosInfo, and DevInfo structures are all linked together according to the scheme shown in Figure 1.0. For that reason, the program first declares four key structure pointer variables in its global-data area. These point to specific instances of the DosLibrary, RootNode, DosInfo, and DevInfo structures; all four pointers are initialized to NULL values.

Once the dos.library is successfully opened, the program obtains pointers to each of the other three structures by reading appropriate pointers in each of these linked structures. The program appropriately typecasts each pointer as it is assigned to the global variables. Notice that Listing.1 also uses the dos.h include file BADDR (BCPL Address) macro to change the rn_Info and di_DevInfo BPTR pointers to normal C pointers.

Just as a matter of curiosity, Listing.1 prints the current addresses of these four structures. Because the Amiga has only one fixed address and these structures are allocated at run time, these addresses may change each time you execute the program, especially if your RAM contents have changed since you last executed it.

The next phase of the program consists of a for loop that looks at the linked list of DevInfo structures. Here it prints key information about all the devices (real, logical, and disk volume) that are currently linked into your system. First, it prints the device name (the dvi_Name parameter) and then the device node type (the dvi_Type parameter). In this for loop the program uses the arp.library BtoCStr function. The BtoCStr changes a BCPL language BSTR string to a C string. If you look at the above definition of the DevInfo structure, you will see that the dvi_Handler pointer is defined as a BSTR pointer. To accommodate this situation, you define a dummy holding string buffer called cString for placing converted C strings. Then in the printf statement where the program prints the dvi_Handler pointer, you reference that C string directly.

The ARP library BtoCStr function returns the number of characters in the resulting C string. Therefore, you print appropriate messages for two different cases to indicate what the presence or absence of a dvi_Handler string really means. (As you saw above, some DOS devices have no handler file in the L: directory.)

The result of this printing loop is a long list of grouped items, each group representing an AmigaDOS device node currently in your system. You will see type 0 (real device) entries for DF0:, DF1:, RAM:, SER:, PAR:, PRT:, CON:, and RAW:; type 1 (logical directory device name) entries for SYS:, EVN:, S:, L:, C:, FONTS:, devs:, and libs:; and type 2 (volume) entries for the disk volume names assigned to the disks currently in DF0:, DF1:, and any other drives in your system.

You can extend the program to determine other key values for all the DOS devices in your system, then use the results to capitalize more fully on you system's resources. ■

Eugene Mortimore is a developer and the author of Sybex's Amiga Programmer's Handbook, Volumes I and II. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

From p. 31

```
##libid Amiga World TJ lib (ver 1.0)
##bias 30 *first function is always at an offset of -30 from lib base
*Here are all of the lib functions callable by an application
*They all return doubles
##ret double
Mag( real, imag )
Ang( real, imag )
Real( mag, ang )
Imag( mag, ang )
##end
```

This one file holds virtually everything you need to describe your library (and update it in the future). Each specialized item has its own command. Of particular interest are the `##init` and `##expu` commands that precede the names of the initialization and expunge routines. If required, commands are available for the previously mentioned open and close routines (`##open`, `##clos`).

COMPILE, LINK, AND GLUE

Using Manx 5.0 from the CLI, you make the library as follows:

```
LibTool -cmho glue.asm AWlib.fid
```

This creates the library startup code (`AWlib.src`) using C-style syntax (it adds an underscore before the function names), the C header file (`AWlib.h`) for the applications, and a C application glue file (`glue.asm`) that will be assembled and linked with the application.

Next, the statement:

```
as -cd -o LibStart.o AWlib.src
```

assembles the library startup module. Because I am using large code and data, I need not worry about saving register A4, as I would with the small model.

Again using the large model, I compile the library code:

```
cc -mcd0b -ff AWlib.c
```

The command suppresses the standard Manx startup (`.begin`), and uses fast floating-point math.

Finally, I create the library by linking together its components.

```
ln -o libs:AW.library LibStart.o AW Lib.o -lmfl -lcl
```

`LibStart.o` always must be the first object module in the list.

To use the library, compile and link your application in the standard manner. If you are using glue files (as I am), assemble `glue.asm`, and then link it with the application program. The example program `AWlib_app.c`, simply opens the library and calls each of the functions.

For information on using `LibTool` with the SAS compiler and assembly development systems, or to create BASIC .bmap files and construct devices, see the on-disk documentation and examples. Whether you use them to house frequently used routines, divide a large program into a series of libraries, or share functions with friends, libraries will make your programming easier and your programs tighter. ■

Jim Fiore is a co-founder of and programmer at dissidents. He has written for computer and music publications and is currently writing an electrical engineering text book. Contact him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458, or on BIX as jfiore.

From p. 33

While we do not see these labels, the drive does, and uses them for file recovery. In both the old and new file systems, the file-list block holds the list of blocks that the file uses.

Our file-recovery programs can still reclaim files as well as ever, but they must work a bit harder than before. Programmers who write file-recovery programs need to allow for the fact that people may use disks formatted under both systems. A recovery program must be able to determine which kind of disk it is reading and supply appropriate branches to the routines needed for each kind of disk.

HACK TO THE FUTURE

The FFS offers substantial improvements over the Amiga's original system. Future changes to the file system will give us much faster access than we now have. Longer file blocks will require fewer reads and writes and less head seeking, which will also make the system faster. With the advent of longer file blocks, we will also see hash tables grow longer than the current 72 slots. The hashing algorithm can then be changed to allow for the bigger tables, and there will be far fewer hash-chain collisions to impede finding files.

The foundation for these changes is already in place. Using such an improved file system will not require the rewriting of programs, but merely that programs address the file system in the established and approved manner. ■

Betty Clay, a sysop on CompuServe's Amiga Forum (76702,337) writes for several Amiga publications. Write to her c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

Perfect Joystick

From p. 35

An equivalent to this is:

```
lea Y,a0
lea X,a1
move.w (a0,d0.w),d1
move.w (a1,d0.w),d0
rts
Y dc.w 0,1,1,0,-1,0,0,-1,-1,0,0
X dc.w 0,0,1,1,0,0,0,1,-1,0,0,0,-1,-1
```

By noting the relationship between Y and X ($X = Y + 22$), you can eliminate one of the LEA instructions, as shown in the first part of this example. You find the vertical position by looking at $Y + D0$ and the horizontal by looking at $X + D0$ (which, because the horizontal table begins 22 bytes after the vertical, is $22 + Y + D0$). Note that both sections look up the vertical value before the horizontal. If I had arranged them in the other order I would have corrupted the value in D0 (the index).

While you probably do not have the time to fine-tune all of your routines, try to occasionally exhaust the possibilities in the search of perfection. The final code I arrived at is quite efficient (see Listing 2 in the accompanying disk's Anderson directory). It comes close to being the best way—in terms of speed and code size—to read the joysticks. ■

Rhett Anderson is co-author of Mapping the Amiga and co-founder of Neandersoft Inc., a game development company. Write to him c/o The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458.

LETTERS

Flames, suggestions, and cheers from readers.

APPLAUSE...

Your idea to create *The AmigaWorld Tech Journal* is great! The demise of *Transactor* was very unfortunate—I bought a Commodore 64 so that I could justify reading the original magazine. For those of us that write software for the Amiga now (whether professionally or for personal enjoyment), a technical journal is extremely important. Even for those who don't program, there are a great many technical issues that need to be covered. Enclosing a disk containing program examples, software tools, program demos, late breaking information, and so forth is also a very good idea. I can't wait!

Richard A. Bilonick
Pittsburgh, Pennsylvania

PROPOSALS FOR SOFTWARE

I have a few ideas for developers. Software should follow certain behavior standards. For example, if the Amiga is a GUI system, then all software should load from the Workbench and have icons, including games. As a further step, get rid of the nonstandard disk operating systems found on some game disks. In the IBM and Mac worlds 99.9% of the software can be put on hard drives. Why shouldn't every application and game on the Amiga include a batch file for copying programs to a hard drive? This brings up another matter; get rid of on-disk copy protection. I favor manual protection, instead.

Chris Browne
Maple Valley, Washington

AND THE JOURNAL...

Please do not ignore the system hardware in your new magazine. The hardware is essentially the Amiga's only feature that distinguishes it from

other computers. Detailed explanations of how the hardware functions and various hacks make for interesting reading.

I also suggest you feature a "clever algorithm of the month" in your magazine. Helping Amiga owners become better programmers instead of game players will help protect the investment we've all made in this machine.

Dan Larson
Shillington, Pennsylvania

PLUS A WARNING

Congratulations on your decision to provide more in-depth technical information to the Amiga community. I know from my own struggles that getting technical details is time consuming and expensive. After several months of floundering around trying to get information, I discovered what a great bargain the Commodore Developer's program is. Whatever you do, do not simply rehash the information that is in *AmigaMail*. Your technical articles should contain "fresh" information that complements what CATS presents in theirs. Please don't misunderstand, I like your idea! A year ago, it would have saved me considerable time and aggravation. I won't pay twice for the same articles, however.

Valorie Jackson King
Bowie, Maryland

DISK DISTRESS

You have stated *The AmigaWorld Tech Journal* will be sold with a disk and implied it will not be available without one. I hope you reconsider this stand and provide a diskless version. While I buy almost every Amiga magazine, I do not purchase magazines with disks. As long as the source to programming articles is provided within the text I am willing and able to input the examples that I need into my editor.

Bob Lockie
Bulington, Ontario

As a former reader of *Transactor*, I was pleasantly surprised to read your announcement of *The AmigaWorld Tech Journal*. I am disappointed, however, that you have chosen to automatically bundle it with software. Please consider offering your readers the option of buying it without the disk. Even if your magazine is very good and a reader has extremely eclectic interests, it is unlikely that more than one or two articles in each issue will be of lasting interest. That means that much of the hefty purchase price will be wasted on programs of little or no value for that particular reader.

Gary A. Gruenhagen
Sierra Vista, Arizona

While I can understand your concerns, we decided to include a disk with every issue to enhance the value of The Tech Journal. By relegating each article's associated source code to disk, we have room in print for more articles and articles on topics that require longer listings than could be printed. As we are also providing a sampling of utilities, libraries, and even a compiler from time to time on the disk, I am confident you'll get your money's worth.

TELL US MORE

How did you like our first issue? Any suggestions for the next? What are your thoughts on the state of Amiga development? Do you have a technical question that you can't find the answer to? Let us know! Drop a note to Letters to the Editor, The AmigaWorld Tech Journal, 80 Elm St., Peterborough, NH 03458. Letters may be edited for space and clarity. ■



YES! enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of \$59.95. That's a savings of \$35.75 off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Canada and Mexico \$74.95. Foreign Surface \$84.95, Foreign Airmail \$99.95. Payment required in U.S. funds drawn on U.S. bank. Available March 15.

Name _____

Address _____

City _____ State _____ Zip _____

☐ Check/Money order enclosed TS411

Charge my: ☐ Mastercard ☐ AMEX ☐ Visa ☐ Discover

Account# _____ Exp. _____

Signature _____

Or call **800-343-0728** ☎ **603-924-0100** for immediate service.

NEW!

6 BI-MONTHLY ISSUES **PLUS** **6 INVALUABLE DISKS**



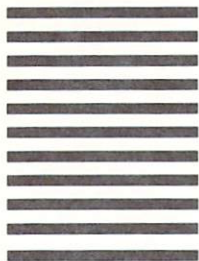
BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT #73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

The AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



The AmigaWorld

TECH JOURNAL

CHARTER
OFFER!

☒ **YES!** Enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of \$59.95. **That's a savings of \$35.75** off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Name _____

Address _____

City _____ State _____ Zip _____

TL411

☐ Check / Money order enclosed

Charge my: ☐ MasterCard ☐ Visa ☐ Amex ☐ Discover

Account# _____ Exp _____

Signature _____

Canada and Mexico \$74.95. Foreign Surface \$84.95. Foreign Airmail \$99.95.
Payment required in U.S. Funds drawn on U.S. Bank. Available March 15.

Or call **800-343-0728** or **603-924-0100** for immediate service.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY MAIL

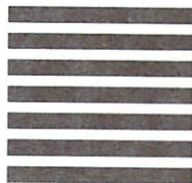
FIRST-CLASS MAIL PERMIT NO. 73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE


AmigaWorld Tech Journal

P.O. Box 802

Peterborough, NH 03458-9971



Sapphire 68020 / 68881 Accelerator

Fits in the Amiga 1000, Amiga 500, and the Amiga 2000. 
Fits snugly in the 68000 socket.

Easy installation - Included is a disk with installation instructions, pictures, and public domain benchmark software. Also included is a static safty strap and static safty instructions that apply to all computer hardware. Factory installed 12 MHz 32-bit 68020 CPU and a factory installed 12 MHz 68881 FPU. Speed increases of up to 2.4 times faster in standard interger processing, and up to 3.2 times faster in floating point. Small compact size makes the Sapphire the smallest accelerator yet! Only 3 1/8" x 4 1/4" x 1/2" total size. Not a psuedo accelerator, but a true 32-bit accelerator card using true 32-bit processors. A full one year factory warranty. Retail Price \$399.95

Workbench Management System v2.0

The Workbench Management System (WMS) is a revolutionary idea in software for the Amiga! WMS is based on a button concept where a single click of the mouse button will launch your applications. Buttons can be assigned to any program on a hard disk, floppy, or network. WMS allows you to launch as many programs as your memory will allow for as fast as you can click the mouse. WMS also includes eight built in programs.

- Memo-ed: A text editor.
 - Calendar: A calendar that you can add daily appointments to.
 - Remind: Is built into the calendar and can be set up to remind you upon boot up every day.
 - Telemate: A phone address book with dail capibiltiy.
 - Squeezebox: Archiving with the compression programs made simple. And More!
- Retail Price \$49.95

RxTools

RxTools is an object oriented user interface builder, which extends the capabilities of ARExx™. It connects the inate textual interface of ARExx™ with the user interface of the Amiga, more commonly known as intuition. Intuition in its raw form is a difficult programming enviroment. RxTools converts intuition into a manageable system with minimal loss of flexibility. Furthermore, it makes it directly accessible in this form to the ARExx™ programmer. With the built in text editor RxTools provides a complete development enviroment, which may be the best available for the Amiga, in terms of overall capability and ease of use. RxTools is a function host program, which when run, stays in the background and allows for the use of the future RxTools extension set to be utilized in ARExx™ scripts. RxTools provides intuition capibility including windows, gadgets, requestors, and more.

Retail Price \$54.95

TTR Development, Inc.

1120 Gammon Ln.
Madison, WI 53719
(608) 277-8071

United States, European, Australian Sales and Distribution

MRBackup Professional

MRBackup Professional is the first full featured backup system for the Amiga utilizing the full potential of the Amiga! With over 60 built in ARExx™ commands, MRBackup Professional gives the user the ability to reach beyond standard backup options. This is the first full featured hard drive backup system that has floppy and SCSI streaming tape capabilities.

- Will backup to floppy or SCSI streaming tape - Fully tested with Commodore, Supra, Xetec, CLtd, Trumpcard, and GVP controllers.
- Full ARExx™ Integration - Over 60 usable commands.
- Utilizes the option to use standard AmigaDOS or Fastdisk Format.
- Has a user selectable file compression - selectable up to 16-bit.
- Uses full AmigaDOS intuition for ease of use.
- Floppy users can use up to 4 floppys.
- MRBackup is compatible with AmigaDOS versions 1.3 and 2.0

Retail Price \$54.95



Teachers Toolkit

Teachers Tool Kit is a complete lesson planner and grade book written by a teacher for the teacher. Teachers Tool Kit allows the instructor to plan the lessons for the class and track the advancement of each student seperatly, or each different class as a whole. Some of the built in features of Teachers Tool Kit Are:

- The ability to store and print graphs, reports, grade book and student information.
- The ability to assign each student a unique ID number.
- The ability to create general subject areas that would have seperate lessons, tests, quizzes, homework, and project that need to be completed by individual calsses.

Retail Price \$49.95



Brigade

A new revolution in gaming software for the Amiga. Most war games work on a turn by turn basis. Brigade takes you one step closer to reality by adding in real time action. Brigade offers excitement not found in other war game simulators. If you do not pay attention or you leave without pausing the game the computer will continue with its war plans and you could lose the game. You may take a break but the enemy never does.

- Real-Time game play.
- Built in scenario/campaign editor - you can create or modify any vehicle, weapon, aircraft, map, and more!
- Oversized map to allow for larger than one screen play.
- Full digitized sound.
- Animation of weapons firing.
- Full control of units, their orders, and missions.
- Special Limited Time Offer - Desert Shield Data Disk Included!!

Retail Price \$44.95

Pre'spect Technics, Inc.

P.O. Box 670, Station 'H'
Montreal, Quebec H3G 2M6
(514) 954-1483

Canadian & South American Sales and Distribution



TAKE THIS JOB AND LOVE IT!

The fastest growing video technology company in the world is looking for technical support specialists. We're looking for the kind of person that enjoys solving problems. If you have the unique ability to: hear a problem, ask the right questions, analyze the answers, and then provide a perfect solution, you belong with NewTek. You'll be a key member of the best support team in the business. Best of all, you'll be working for a company that has an intense commitment to serving its customers.

We're looking for rare individuals with as many of the following qualities as possible:

- Excellent written and verbal communication skills
- Experience in video production and editing
- Knowledge of 3D animation and computer graphics

- Knowledge of Amiga software and peripherals
- Experience with Digi-View and Digi-Paint
- Experience with the Video Toaster
- Work experience in the technical support field

The Video Toaster is changing the way the world thinks about video production. The Toaster is being used everywhere from networks to cable stations and our users range from rock stars to wedding videographers. Helping producers, artists, and video makers use this powerful tool to revolutionize video is an exciting and rewarding career. If you're ready to make a change for the better call Kiki Stockhammer at 913-354-1146.

215 S.E. Eighth St.
Topeka, KS 66603
913-354-1146
FAX: 913-354-1584

NEWTek
INCORPORATED